From Hardware Verification to Software Verification
Re-use and Re-learn

Haifa Verification Conference
October 25, 2007

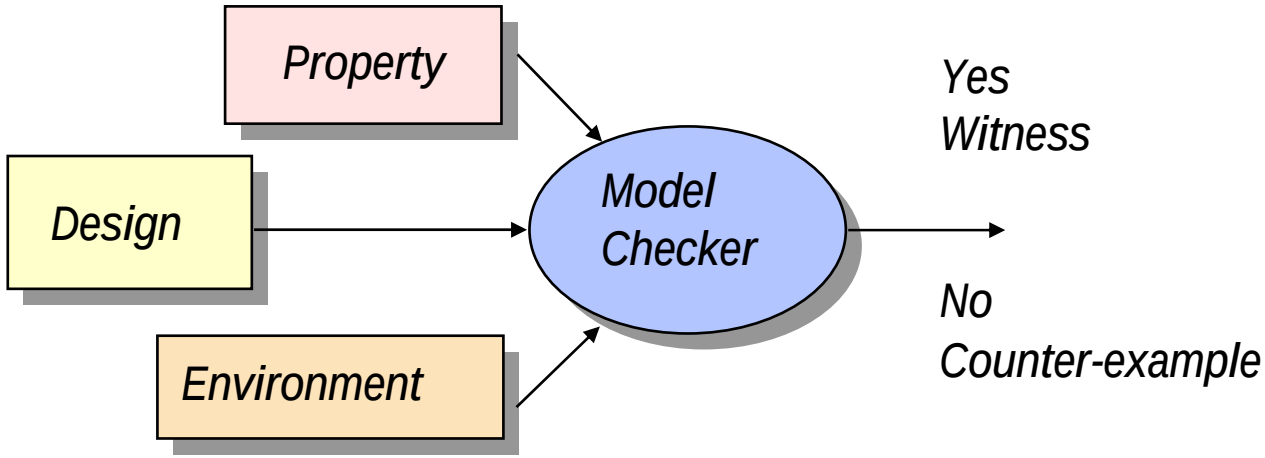
Aarti Gupta
agupta@nec-labs.com
NEC Laboratories America
Princeton, USA

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

Outline

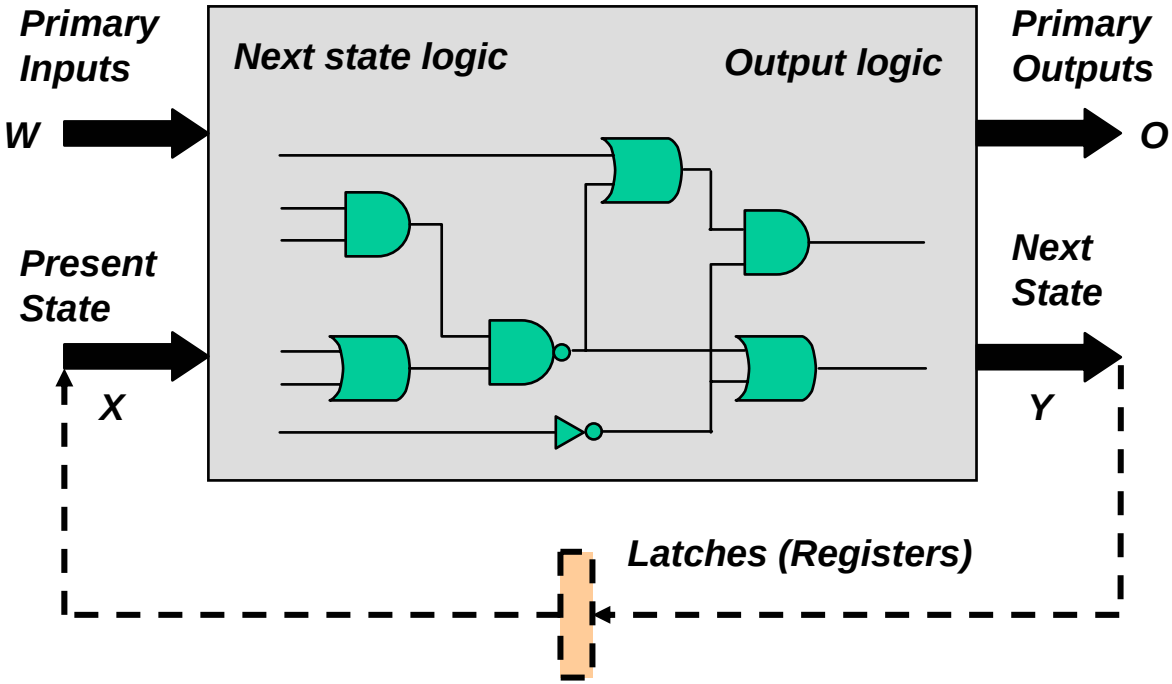
- **Background**
  - Model checking
  - Hardware verification
  - Software Verification
- **Re-use & Re-learn**
  - Symbolic model checking
  - Exploiting high-level structure
  - Supplement (with a little help from *cheaper* friends)
  - Verify at intermediate functions
  - Model transformations
- **Practical Experience**
- **Lessons Learned**
- **Challenges**

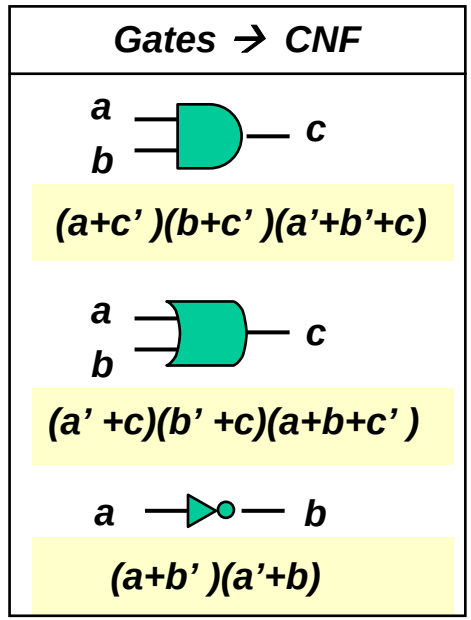
Model Checking (MC)



- **Model checker: Checks whether the design satisfies the property by exhaustive state space traversal** [Clarke *et al.* 82]
- **Advantages**
  - **Automatic verification method**
  - **Provides error traces for debugging**
  - **No test vectors required: all inputs are automatically considered**
  - **Sound and complete (no false proofs, no false bugs)**
- **Practical Issues**
  - **State space explosion (exponential in number of state elements)**
  - **The system needs to be closed**
    i.e. we need to model the environment (constraints on design inputs, or models)

Symbolic Hardware Circuit Model

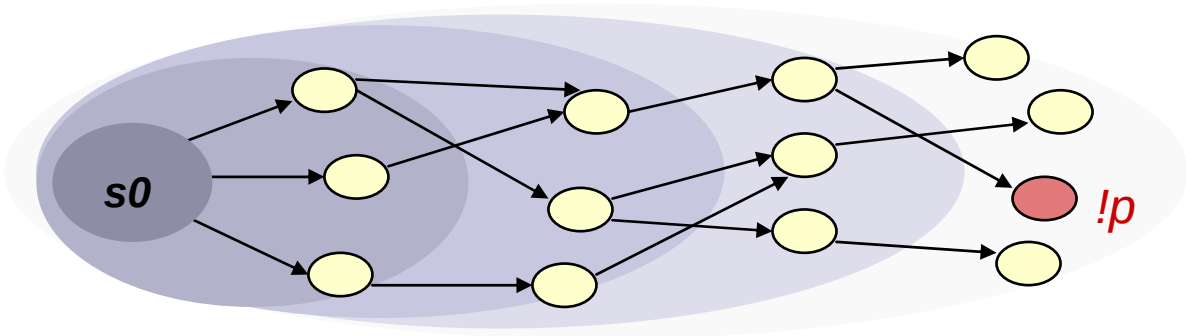




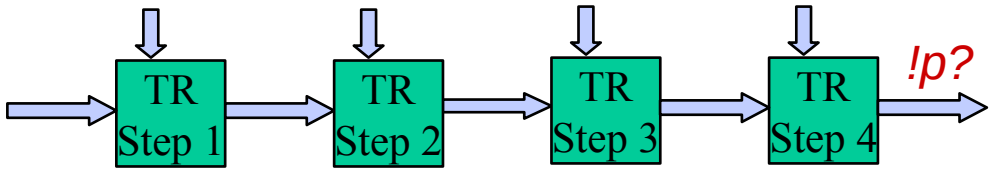
- Design is modeled as a Labeled Transition System (LTS): (S, s0, TR, L)
- Symbolic LTS Representation
  - Set of States S is encoded by a vector of binary variables X (outputs of latches)
  - Initial state s0 comprises initial values of the latches
  - Transition relation TR is implemented as next state logic (Boolean gates)
    - Can also be represented in Conjunctive Normal Form (CNF)
  - Labeling L is implemented as output logic (Boolean gates)
- Note: Size of state space S = $2^{|X|}$

Property Verification

- **Proof Approach: Model Checking**
  - Exhaustive state space exploration
  - Maintains a representation of visited states (explicit, BDDs, circuit graphs)
  - Very expensive for medium to large-size LTS
- **Falsification Approach: *Bounded* Model Checking**
  - State space search for bugs (counter-examples) only
  - Typically does not maintain representation of visited states
  - Less expensive, but needs good search heuristics



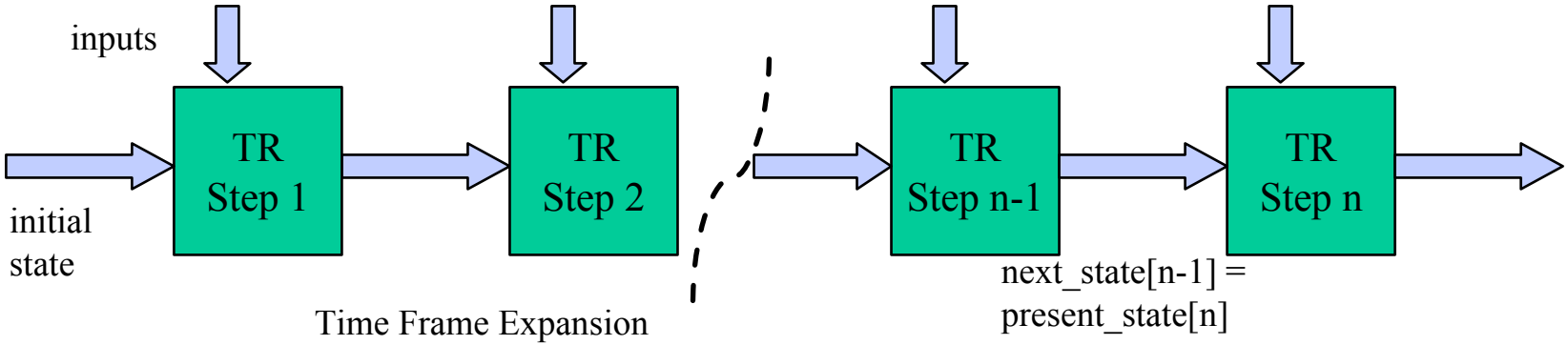
Model Checking AGp
Does the set of states reachable from s0 contain a bad state(s)?



Bounded Model Checking
Is there is a path from the initial state s0 to the bad state(s)?

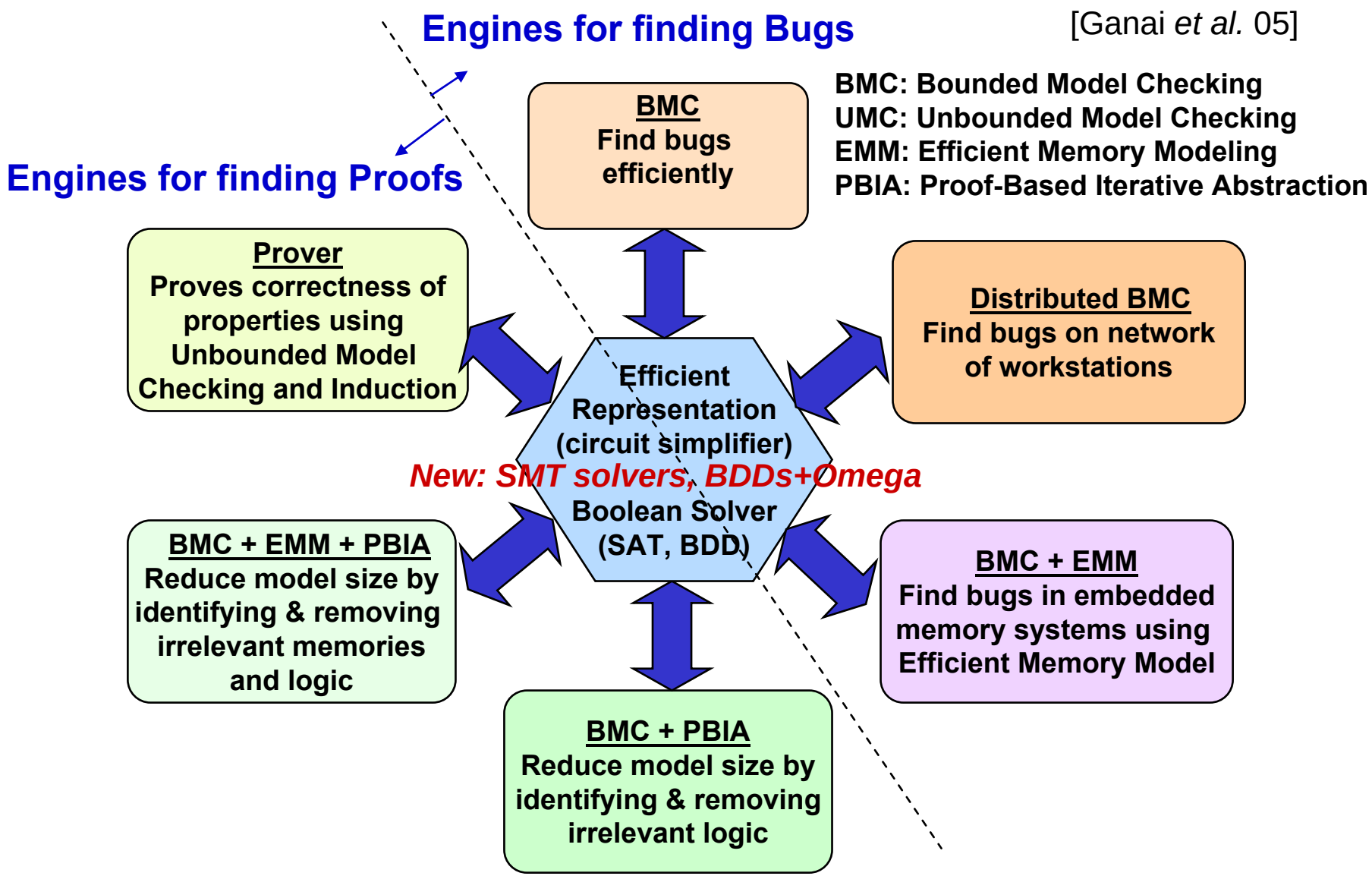
Bounded Model Checking (BMC)

- **Main idea: Unroll transition relation logic up to *bounded length***



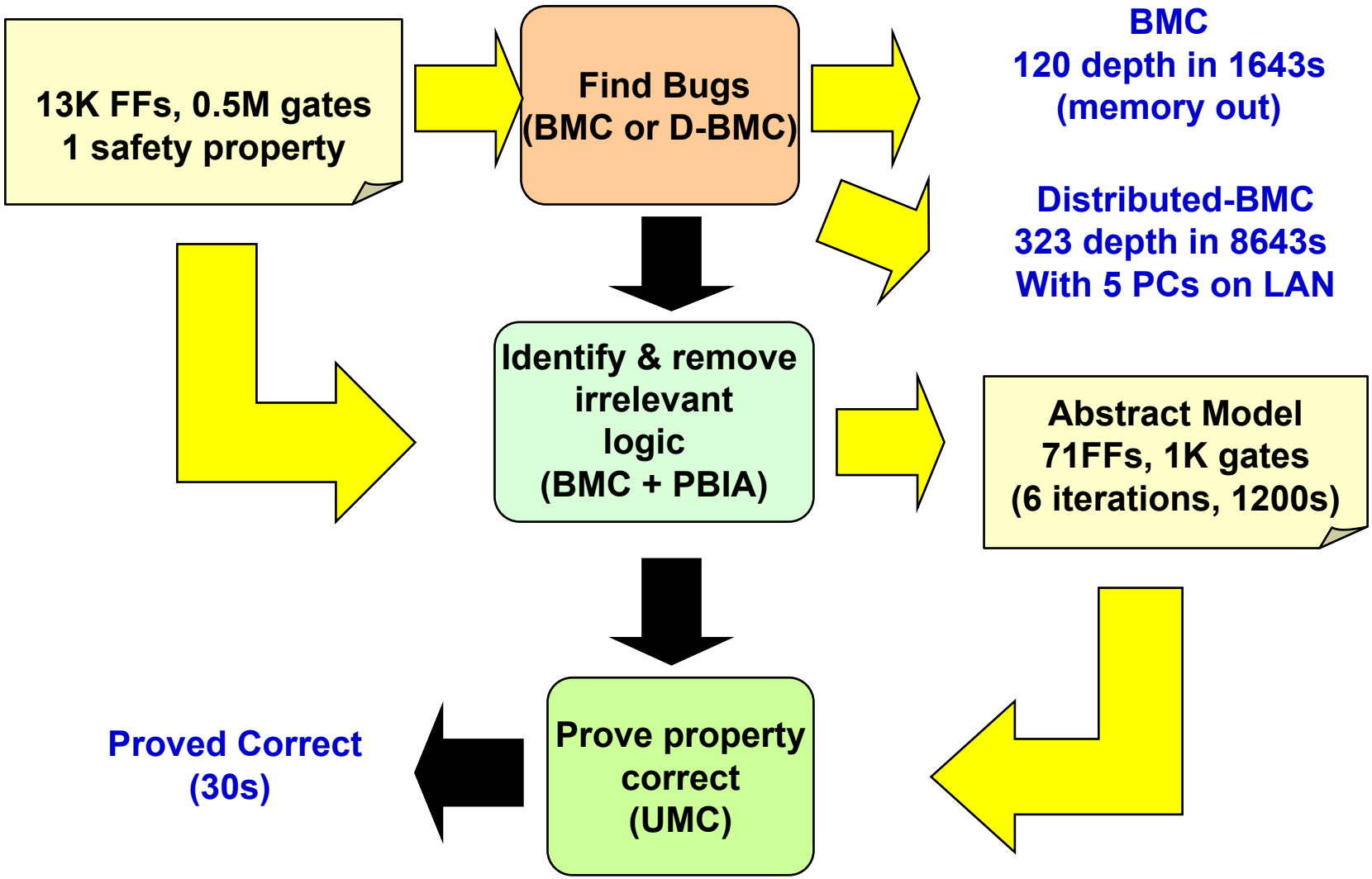
- **BMC problem translated to a Boolean formula *f*** [Biere *et al.* 00]
  - **A bug exists of length *k* ⇔ *SAT(f_k)* (formula is satisfiable)**
  - **Satisfiability of f_k is checked by a standard SAT solver**
- **Falsification: Can check for bounded length bugs**
  - **Scales much better than symbolic model checking with Binary Decision Diagrams (BDDs)**
  - **BDDs: 100s of latches, SAT-based BMC: 10k of latches**
- **Proofs by induction with increasing depth** [Sheeran *et al.* 00]
  - **Works well with additional reachability invariants** [Gupta *et al.* 03]
- **Proof-based abstraction for unbounded MC** [McMillan & Amla 03, Gupta *et al.* 03]
- **Interpolant-based model checking** [McMillan 03, McMillan 06]

VeriSol Symbolic Model Checking Platform



Industrial Case Study: Multiple Verification Engines

Interesting large problems are within reach!



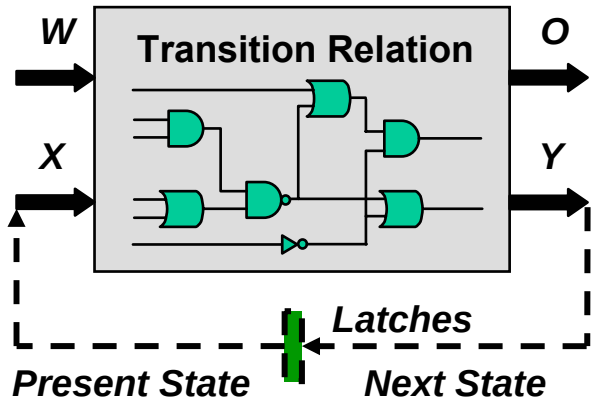
Re-use Symbolic Model Checking Platform for SW

C Program

```
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: }
```

Huge gap !

Symbolic LTS Model
M = (S,s0,TR,L)



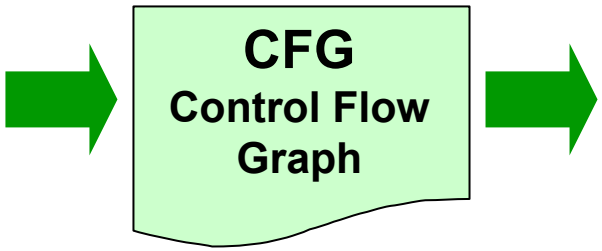
Challenges:

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

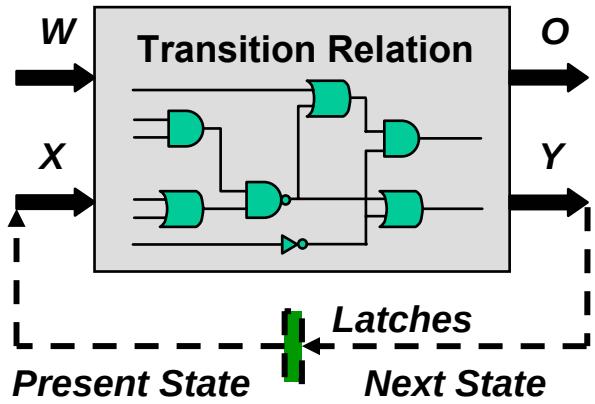
Intermediate Representation

C Program

```
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: }
```

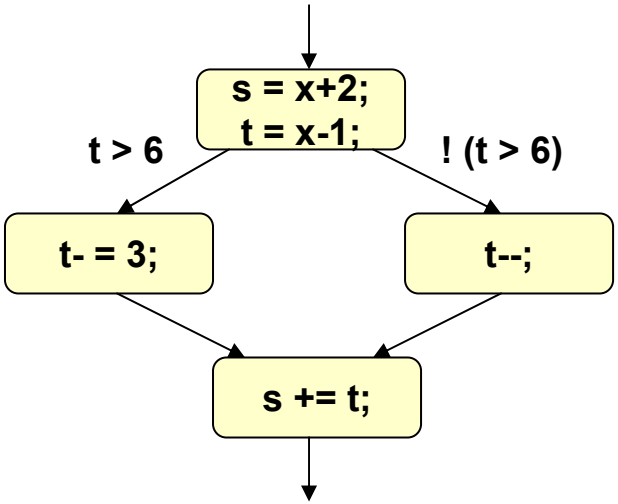


Symbolic LTS Model
M = (S,s0,TR,L)



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

Automatic Translation of CFG to Symbolic LTS



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

Re-use: Bit-level accurate models for precision (e.g. 32-bit adders, shifters, etc.)

- **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**
    - Similar to work in high-level synthesis [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 systems, Boolean programs [Ball & Rajamani 01]
  - **Concurrent programs: Each thread is represented by a separate CFG, with shared variables**

Outline

- ✓ Background
  - ✓ Model checking
  - ✓ Hardware verification
  - ✓ Software Verification
- ❑ **Re-use & Re-learn**
  - **Symbolic model checking**
  - **Exploiting high-level structure**
  - **Supplement (with a little help from *cheaper* friends)**
  - **Verify at intermediate functions**
  - **Model transformations**
- ❑ Practical Experience
- ❑ Lessons Learned
- ❑ Challenges

Symbolic Model Checking of Software Models

- **Back-end verification is performed by the VeriSol platform**

- **Falsification: Bugs (reachability of error labels) can be found by using SAT-based BMC on the software models** [Ivancic *et al.* 04]
  - **Unrolling of TR corresponds to a block-wise execution on the CFG**
  - **SAT-based search relatively insensitive to number of variables**
    - Due to effective conflict-driven learning and related decision heuristics
  - **Difference from HW: Depth of unrolling required to reach bugs may be very high (need to start from `main()`?)**

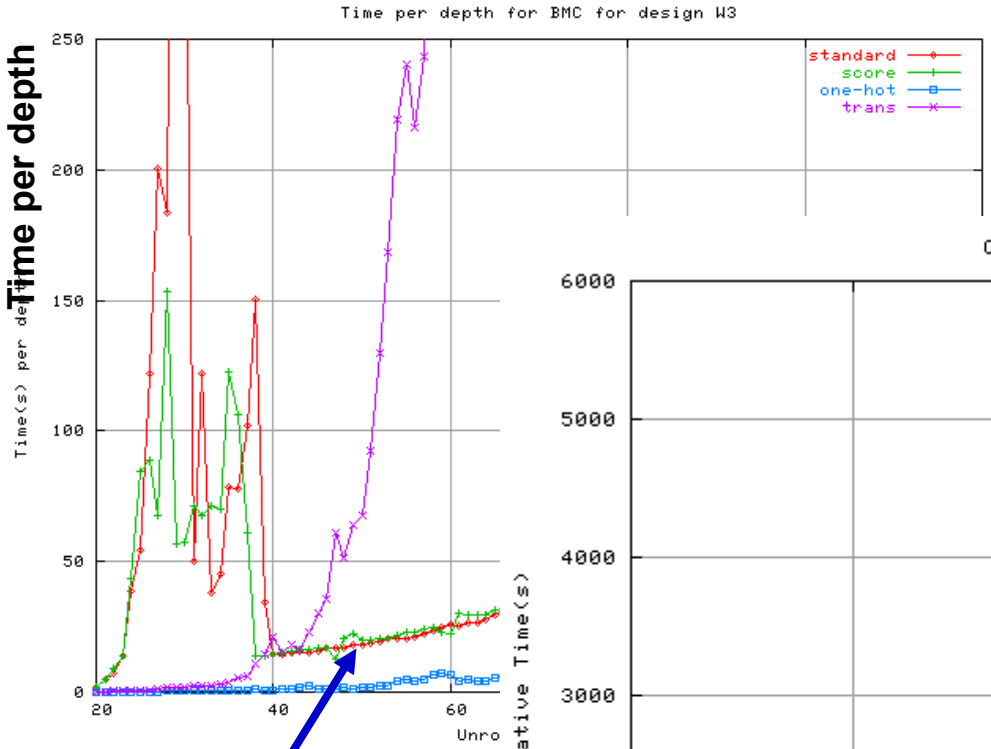
- **Verification: Proofs can be derived by using SAT-based or BDD-based unbounded model checking on the software models**
  - **Methods that save sets of reachable states tend to blow up**
  - **Difference from HW: number of state variables in model is much larger (number of `int` variables * 32?)**

Re-learn: Exploit Structure of SW models

- **Use customized SAT heuristics for SW models** [Ivancic *et al.* 04]
  - **Observation: Only a single basic block is active in a sequential program**
    - **Use a binary-encoded program counter variable (rules out other values)**
  - **Observation: Program control location determines values on data**
    - **Heuristic pc: Make decisions on "program counter" variables first (give them a higher score than other program variables)**
    - **Heuristic one-hot: Allow word-level decisions on the program counter, by using a one-hot encoding (e.g. B5 = (pc == 000101))**
  - **Observation: Each basic block has relatively few predecessors**
    - **Heuristic pred: Add (redundant) predecessor constraints to prune search**
  - **Observation: Each basic block has relatively few successors**
    - **Heuristic succ: Add (redundant) successor constraints to prune search**

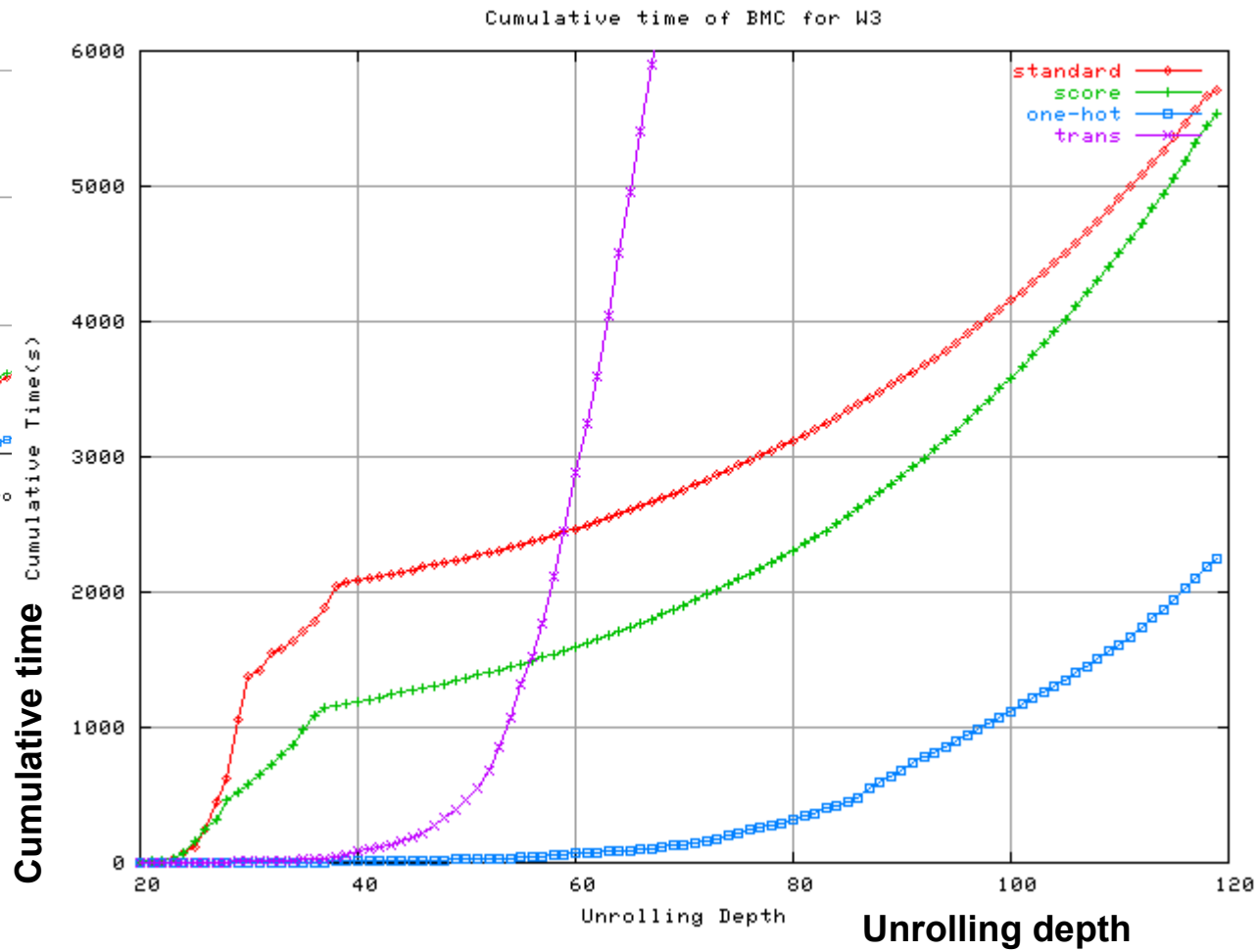
- **For BDD-based model checking, exploit the sequential nature to use a disjunctively decomposed MC algorithm** [Wang *et al.* 06]
  - **Faster and more memory-efficient than conjunctive BDD-based MC**

Results for PPP Case Study



Learning in SAT

- standard SAT heuristics
- higher score for pc
- one-hot encoding of pc
- predecessor constraints



Re-learning Strategies

- **Issues so far**
  - **Bit-accurate models from CFGs may be too large**
    - **even after property-based program slicing**
  - **Depth of unrolling required may be too high**
- **Strategies**
  - ✓ **Use customized heuristics for SAT or BDDs**
  - ✓ **Use light-weight analyses on CFGs to reduce model size**
    - **Range analysis for bounding #bits per variable**
      - For example, `for (i=0; i<10; i++)` does not require 32 bits to represent `i`
    - **Constant folding**
      - Very effective for our memory modeling, which assigns locations to variables
  - ✓ **Use predicate abstraction and refinement** [Slam, Blast, CBMC/Satabs]
    - **Despite localization techniques, this frequently blows up** [Jain *et al.* 05]
    - **Does not work well on programs with pointers**
  - → **Use cheaper static analysis methods to supplement model checking**
    - **Static invariant generation**
  - → **Verify starting from intermediate functions, not entire program from `main()`**
    - **More scalable**
  - → **Use transformations to generate "verification friendly" models**
    - **Provides 1-2 orders of magnitude performance improvement**

Supplementing Model Checking: Motivation

```
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 ;
}
```

Checkers inserted

```
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:
$0 \le i \le N$
$0 \le j \le N$
i==j

- **Example**
  - **No error possible because of invariants that hold true at if-statements**
  - **Difficult to prove by model checking (large model for large N)**
  - **Difficult to prove by predicate abstraction refinement**
    - Weakest pre-condition based refinement cannot discover the relationship i==j
- **Such invariants can be easily (cheaply) discovered by static analysis**
  - **e.g. by using Octagon abstract domain**

Static Invariant Generation

- **Octagon abstract domain: ± x ± y ≤ c**
  - Due to Antoine Miné
  - Successfully used in ASTRÉE static analyzer
  - Captures commonly occurring variable relationships
    - Array bound accesses
  - For n variables, uses $O(n^3)$ time and $O(n^2)$ space to find invariants
- **More expressive abstract domains can be used**
  - Linear invariants: more expensive

- **Applications in SW verification**
  - Invariants can prove correctness of many properties [Sankaranarayanan *et al.* 06]
    - Array buffer overflows, null pointer dereferences, …
    - Acts as a filter: reduction in # properties passed to the model checker
  - Invariants can aid predicate abstraction refinement [Jain *et al.* 06]
    - Improve performance by reducing number of refinements
  - Invariants can aid bounded model checking [Ganai *et al.* 06]
    - Improve performance by pruning SAT search space

Results on Industry Programs

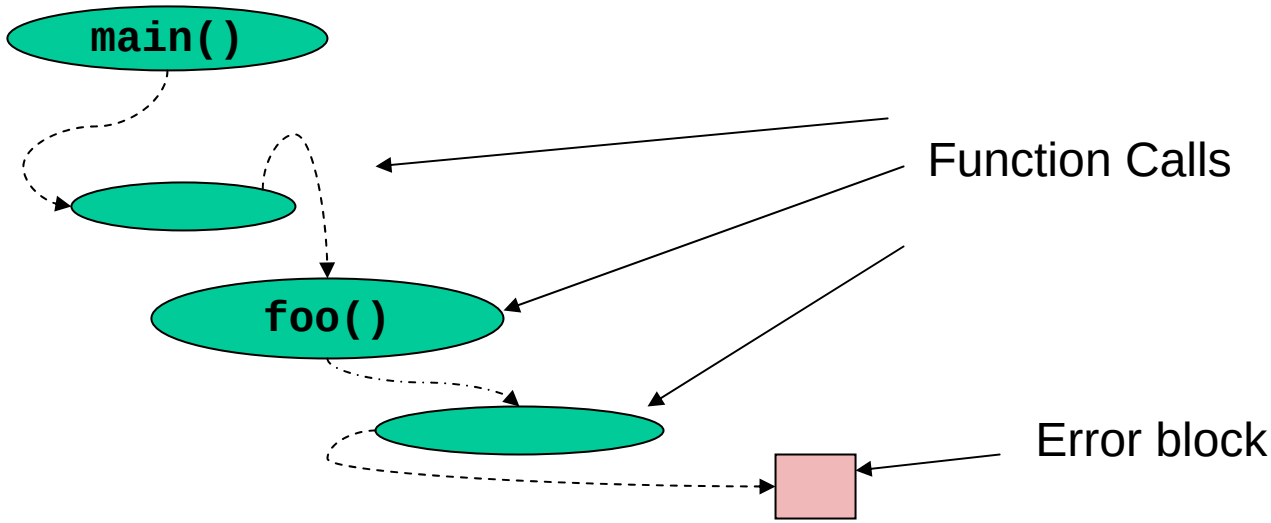
- **Octagon invariants**
  - Can find many proofs of correctness before invoking the model checker
  - Provide overall performance improvement

			Without Octagon Invariants					With Octagon Invariants				
	KLOC	# Buffer overflow checks	# P by SA	# P by SAT	# B by SAT	# Inc.	Time (sec)	# P by SA w/ Invar	# P by SAT	# B by SAT	# Inc.	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, Inc. = Inconclusive

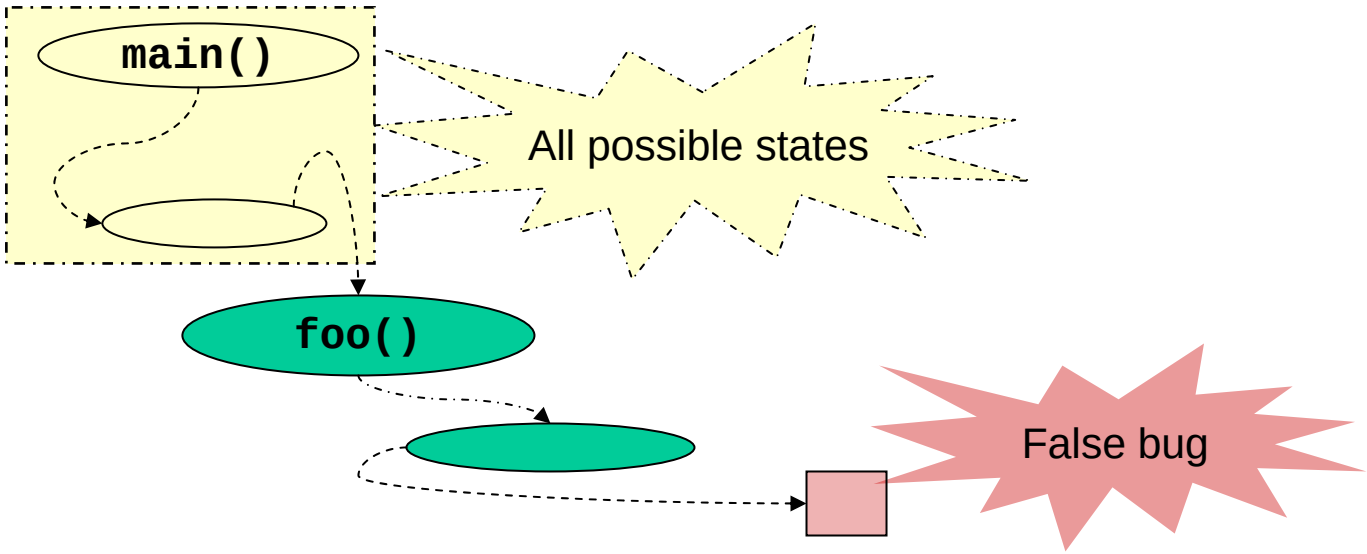
Verifying Programs from Intermediate Functions

- **Ideally, deep bugs starting from the `main` function can be found by MC**



- **In practice**
  - **Model checker runs out of time/memory**
  - **Missing function calls, global initialization**
  - **`main` function contains parsers (deep recursion)**
- **Bugs can be found at a smaller depth starting from an intermediate function `foo(..)`**
  - **Re-use: Similar to Localization Reduction** [Kurshan 94]
  - **Issue: How to supply the context, i.e. the environment for the entry function?**

Conservative Environment Abstraction



- **Start from an intermediate function, with a conservative environment model**
  - **For example, input parameters can take any values**
  - **For example, pointer parameters/globals can be aliased or non-aliased**
  - **For example, input parameters are independent (no relationship), etc.**
- **Many reported bugs will likely be *false bugs*, due to missing context**
- **Reuse: Use these counterexamples to refine the environment iteratively**
- **CEGER: CounterExample Guided Environment Refinement**
  - **Specialized CEGAR: CounterExample Guided Abstraction Refinement** [Clarke *et al.* 00]

Example of Environment Model

```
int * x;
void main (int n)
{
  int  i;

  if (n < 0 || n > 1024 )
       return;
  x = malloc(size(int)*n);
  if (!x) return;
  init_array(x+n);
  . . .
}

void init_array (int *y)
{
   int *j;

   for ( j=x; j<y; ++j)
   {
      *j = 0; //error check
   }
   return;
}
```

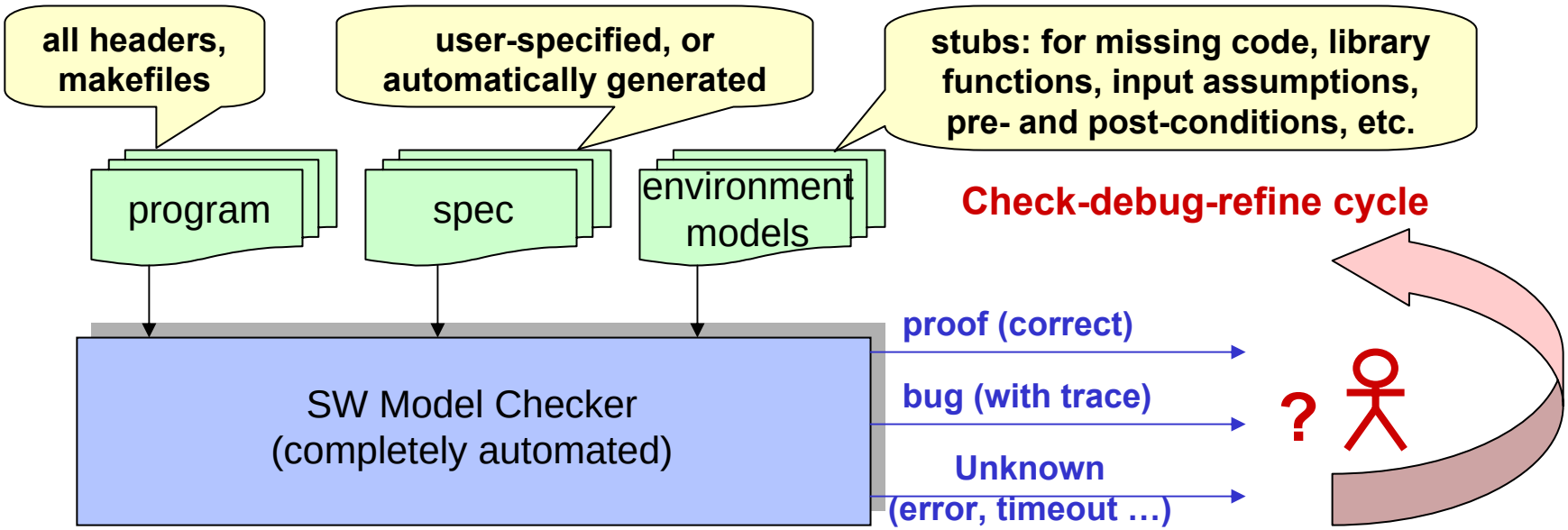
- Verification with main( ) as entry
  - No bugs reported for array buffer overflow or null pointer dereference at error check
- Verification with init_array(..) as entry
  - (Bogus) Bug reported
  - Missing context: Relationship between x and y
- A possible solution: Use the following stub

```
void init_arrays_env ( int * y)
{
      env_assume(x);
      env_assume(y);
      int k = env_get_array_bound(x);
      env_assume( y > x && y <= x + k );
}
```

- Model checker can automatically reason with an (iteratively refined) stub for each entry function
- Similar technique is successful in finding the well-known array overflow bug in bc-1.06 (gnu)

CEGER: CounterExample Guided Environment Refinement

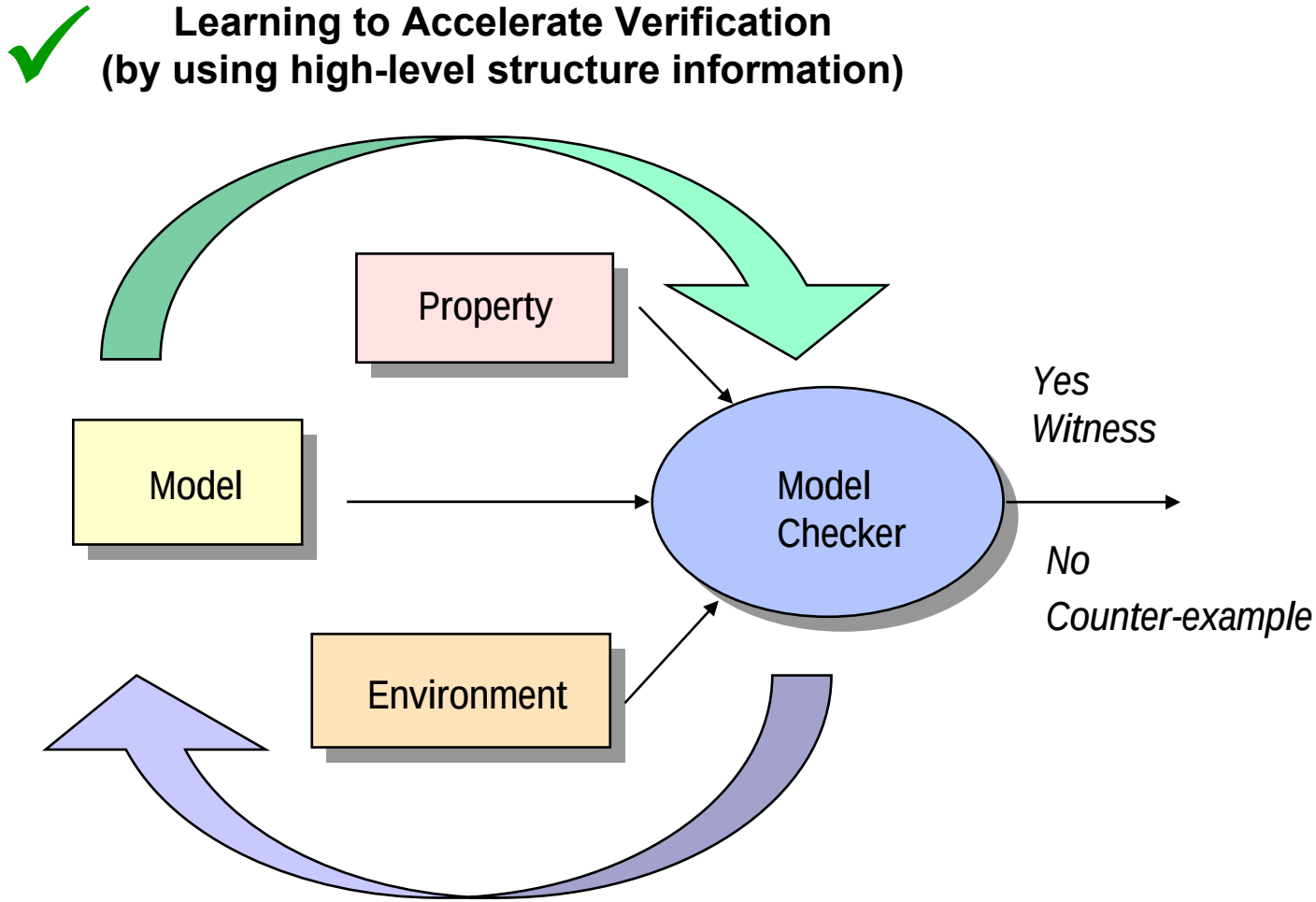
- **Not completely automated: User in the loop**
  - To determine whether a reported bug is false
  - To provide the environment refinement
- **However, automated help is available from the model checker**
  - Counterexample trace (exhaustive/bounded search is the hard part)
    - Users can examine the interface variables of interest and generalize relationships
  - Projection of counterexample trace on the interface (weakest preconditions, interpolants)
    - Users find it easier to modify a suggested constraint, than to come up with one
- **The overall flow is consistent with a check-debug-refine cycle**



CEGER for SW Verification

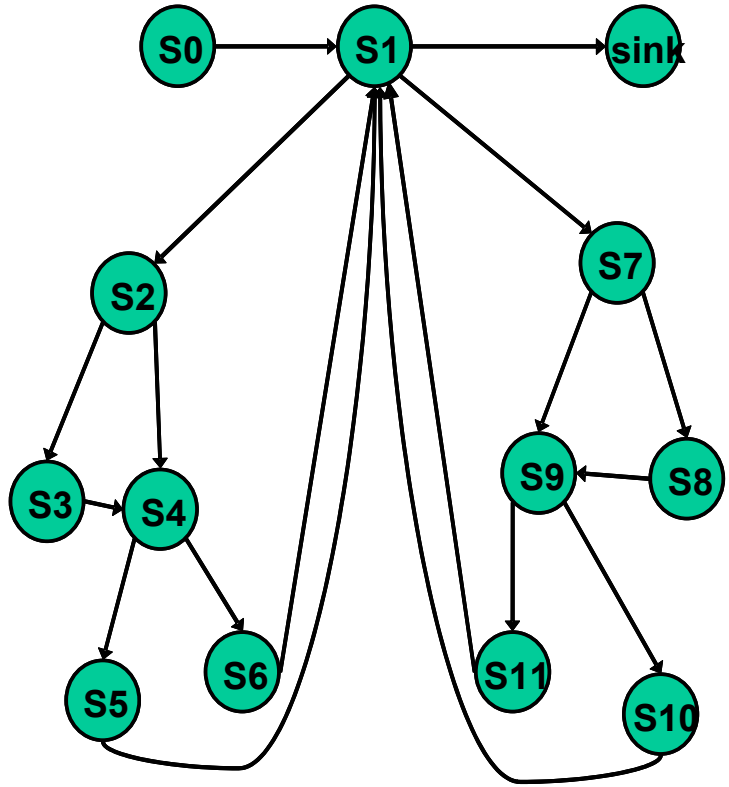
- Modular verification by utilizing interface constraints is more scalable
- Differences between HW and SW components
  - Nature of composition
    - HW: typically synchronous composition
    - SW: typically asynchronous composition (wrt atomic blocks/functions)
  - Effect of primary inputs
    - HW changes state in every cycle in response to primary inputs.
    - In SW, most state changes internal to a function do not depend on primary inputs. The inputs typically affect only the starting state of a multi-step (atomic) function.
  - Type of interface constraints
    - In HW, may need to capture sequences of constraints on interface signals
    - In SW, typically Hoare-style pre-conditions/post-conditions suffice
  - Number of primary inputs at component boundaries, relative to component size
    - Much larger in HW, than number of input parameters of functions in SW
- CEGER is likely better-suited for SW than for HW
- (Automatic) Generation of SW component interfaces
  - Learning methods, Interface automata, Refinement-based methods …

Exploiting High-level Structure (Again)

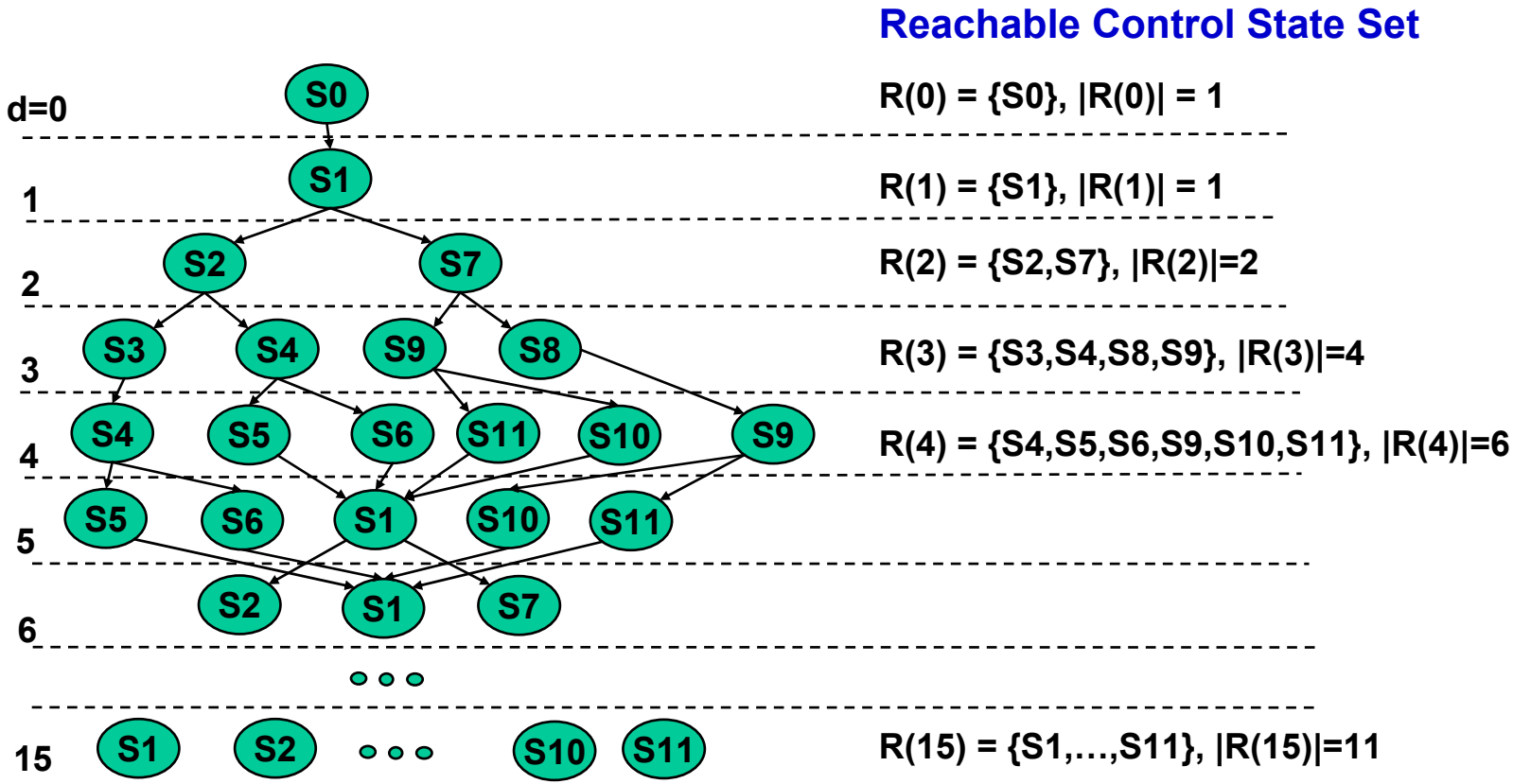


Model Simplification

- **HW model simplification results in significant performance improvement in BMC**
  - Circuit graph hashing, BDD-sweeping, SAT sweeping [Kuehlmann *et al.* 00, 01, 04]
- **Re-use: Simplify the CFG to improve BMC**
- **Example CFG**
  - The only control states reachable at depth 3 are S2, S7 (and sink)
  - Pruning of search space is possible by prohibiting all other control states at depth 3
  - These constraints are propagated for further reduction in size of BMC problem
- **Control State Reachability Analysis (CSR)**
  - Conservative reachability analysis on control states only (without regard to guards)
  - Provides useful constraints in BMC to help size reduction and search space pruning



Control State Reachability Analysis: Limitation

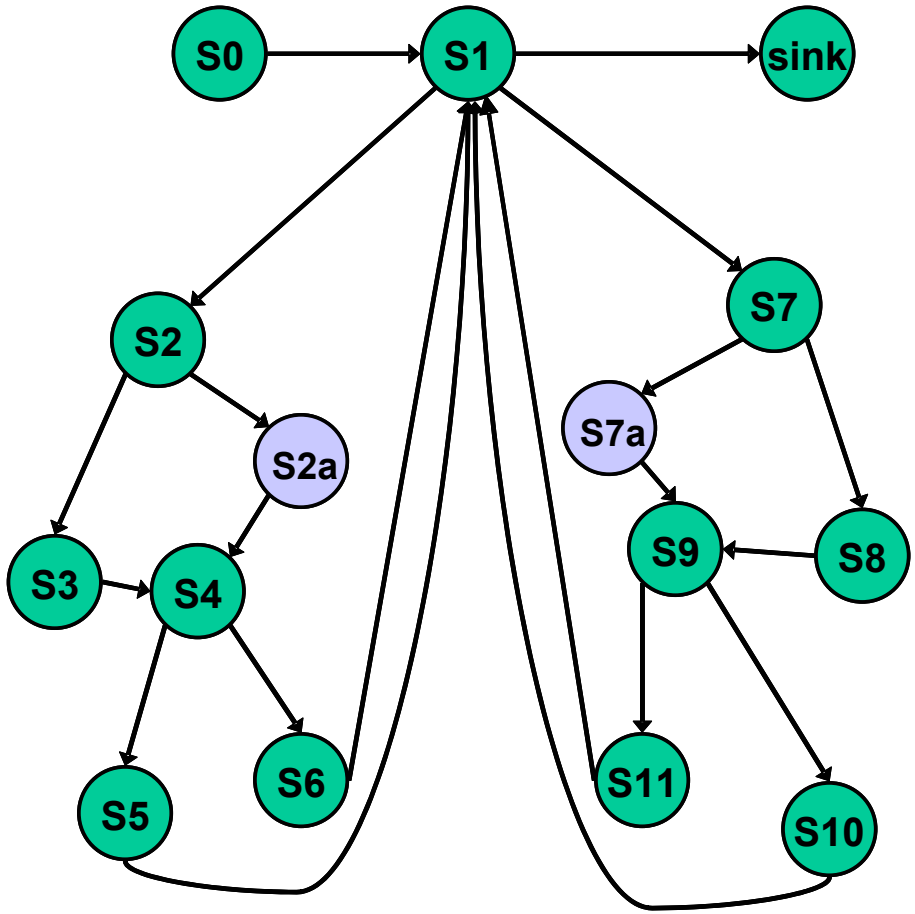


- **Problem: All control states may become reachable after some depth**
  - Example: No search space pruning in BMC after depth 15
- **Why does this *Saturation* happen?**
  - Mainly due to reconvergent paths in the CFG that have different lengths
  - This is especially problematic in reconvergent paths inside loops

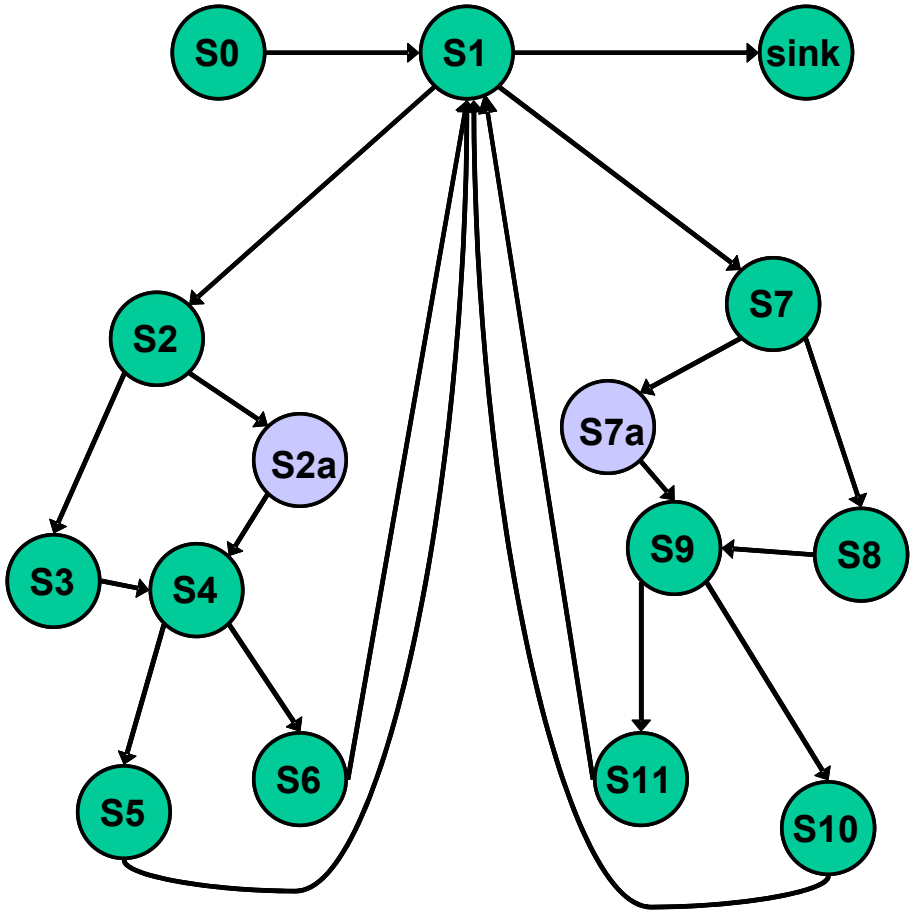
Model Transformation: Path Balancing

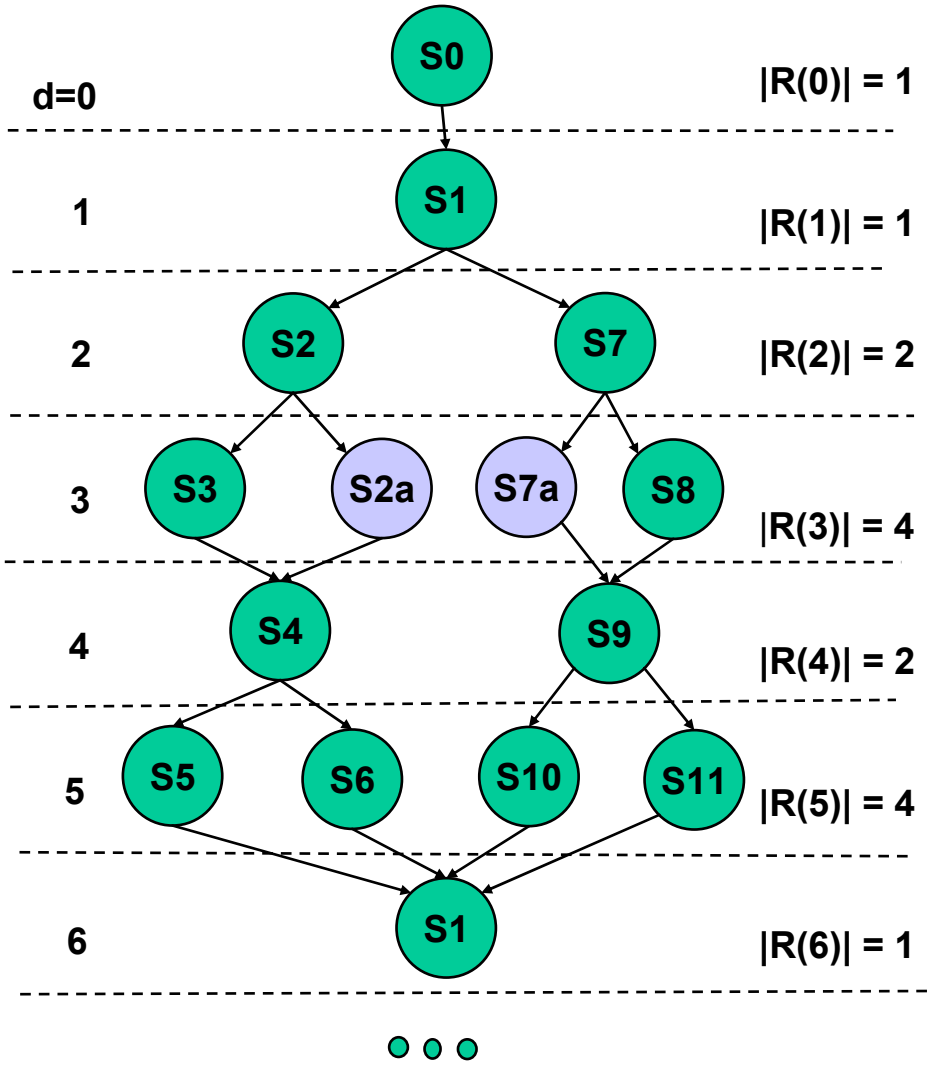
[Ganai & Gupta 06]

- **Re-learn: *Transform* the CFG to improve BMC performance**
- **Path Balancing Strategy**
  - Balance reconvergent paths in CFG by inserting NOP states
  - This increases path lengths, but …
  - Potentially provides benefits in BMC
- **Example**
  - Add states S2a and S7a (NOP states)
  - Preserves typical properties of interest (without "Next-time")



Control State Reachability on Path-Balanced CFG





No Saturation in CSR !

Max |R(d)| = 4

Provides search space pruning and size reduction in BMC at all depths

Results: Learning + Model Transformation

- Industry software C-program: 17 KLOC
- 6 properties P1-P6, checked using SMT-based BMC
- M → M' : Model transformation (Path Balancing)

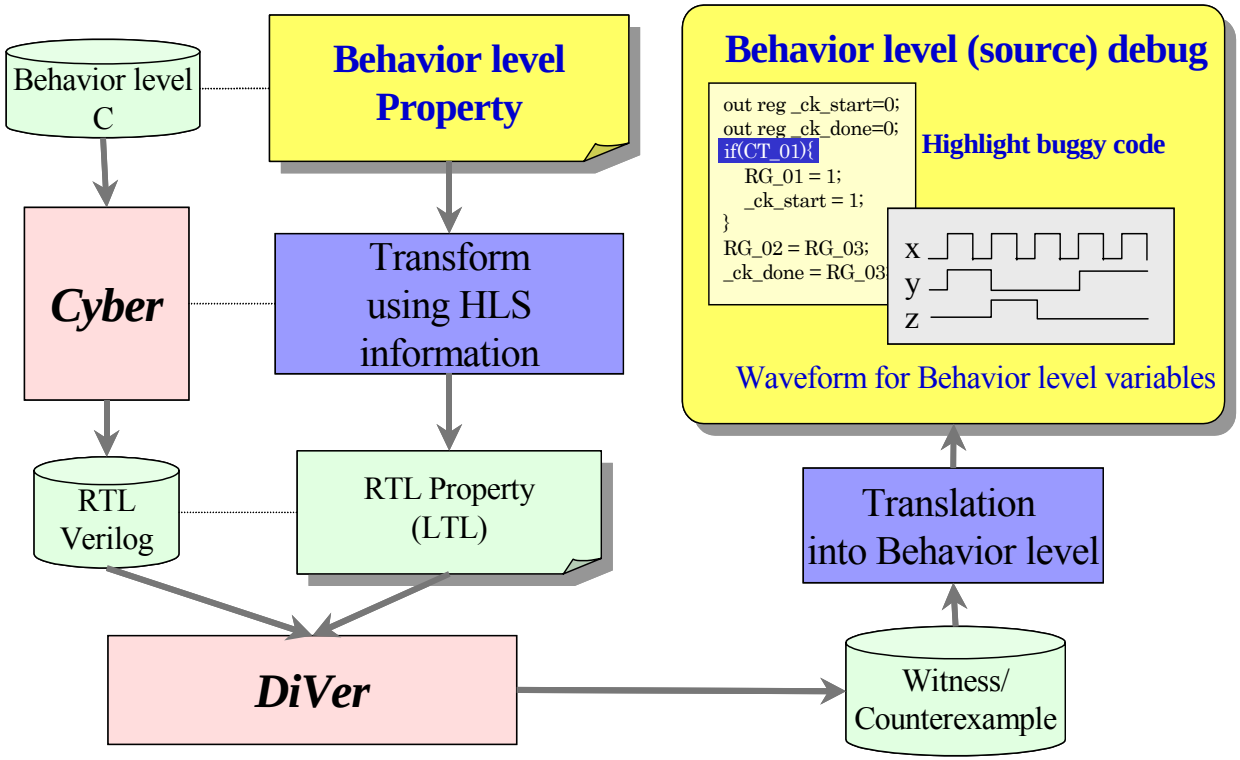
P	A: Without Learning on M			B: With Learning on M			C: With Learning on Transformed M'		
	D	sec	Wit?	D	sec	Wit?	D	sec	Wit?
P1	9*	TO	No	38*	TO	No	41	<1	Yes
P2	9*	TO	No	41*	TO	No	44	<1	Yes
P3	9*	TO	No	43*	TO	No	92	156	Yes
P4	9*	TO	No	30	188	Yes	94	151	Yes
P5	9*	TO	No	21	6	Yes	60	4	Yes
P6	9*	TO	No	31	164	Yes	70	22	Yes

Notes: D: Analysis Depth (denotes depth at timeout 1000s), Wit: Witness found? Yes / No*

Outline

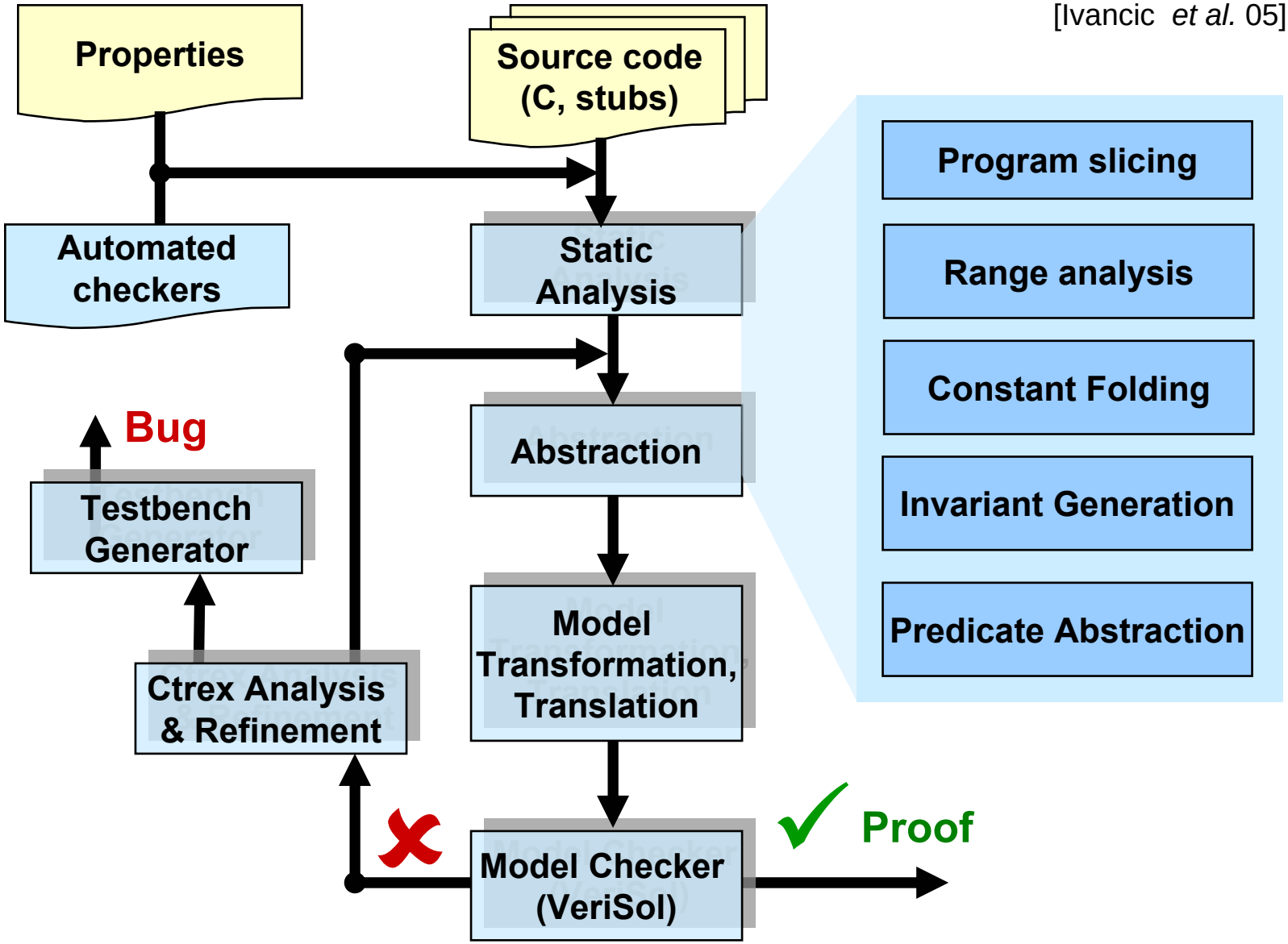
- ✓ Background
  - ✓ Model checking
  - ✓ Hardware verification
  - ✓ Software Verification
- ✓ Re-use & Re-learn
  - ✓ Symbolic model checking
  - ✓ Exploiting high-level structure
  - ✓ Supplement (with a little help from *cheaper* friends)
  - ✓ Verify at intermediate functions
  - ✓ Model transformations
- ❑ **Practical Experience**
- ❑ **Lessons Learned**
- ❑ **Challenges**

HW Verification in High Level Synthesis Framework



- **Cyber Work Bench (CWB)**
  - Developed by NEC Japan (Wakabayashi *et al.*)
  - Automatically translates behavior-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

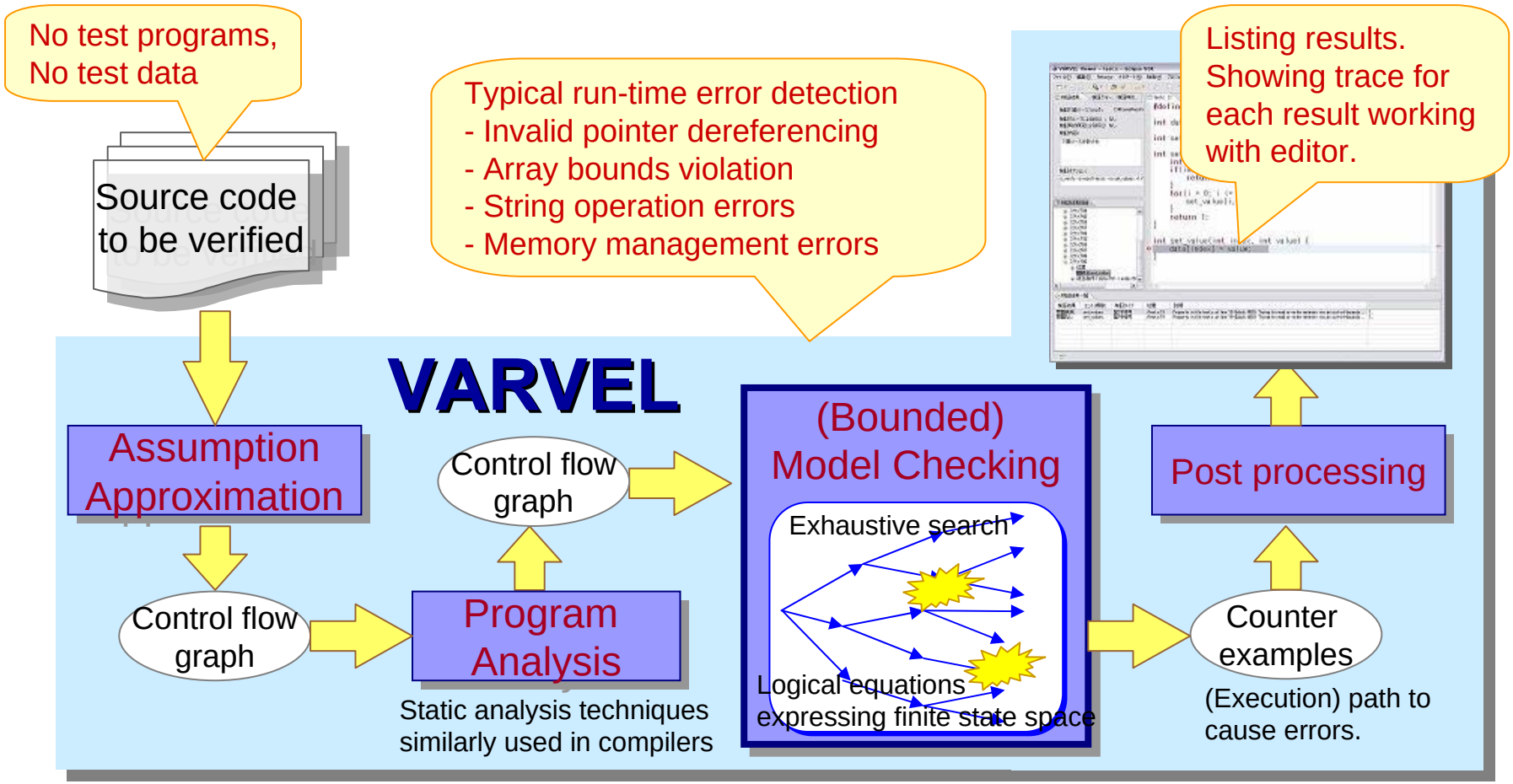
F-Soft Software Verification Platform



VARVEL: Source code Verification Tool for C

Acknowledgement: Y. Hashimoto *et al.*, NEC

- Statically detects typical run-time error for C from source code
- With bounded model checking, check variable values and paths accurately
- Currently in practical in-house use for commercial product software



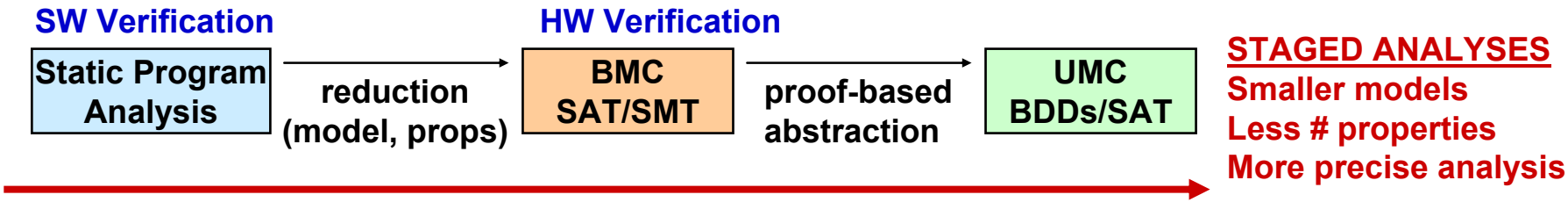
In-house Verification Service

Acknowledgement: Y. Hashimoto *et al.*, NEC

- **NEC Japan has started in-house "Source code Verification Service"**
  - Service provided by SW developers, not verification experts
- **Verified source code of several commercial SW projects**
  - Total lines : about 1.8 MLOC (up to 600 KLOC in one project)
  - Verified source code had been already tested
- **Detected 37 confirmed bugs (500+ potential bugs)**
  - Projects determined the service is effective.
  - The verification service will be incorporated into software development process in divisions of those projects.

Lessons (Re-)Learned

- **Accuracy of program modeling *AND* efficiency of analysis are crucial**
  - Conflicting requirements in general (knob can be tuned for specific properties)
  - Much harder to regain global accuracy from local reasoning (e.g. in predicate abstraction refinement)
  - Retain bit-level accuracy (in some model), spend effort on improving analysis
- **Advancements in constraint solvers (SAT, SMT) offer hope**
  - Sophisticated search heuristics and learning are useful for finding bugs
  - More scalable than methods that save states
- **Exploit the structure in SW models (vs. flat HW-like models)**
  - Learn from models to solver, from solvers to model
- **Stage the analyses (cheaper methods first!)**
  - Difficult to handle MLOC, 1000s of properties: no silver bullet
  - Stage the analyses to reduce model and # properties, and to improve precision
  - Pay attention to proofs (if only to avoid wasting time to find bugs)



Challenges & Future Directions

- **Pointer alias analysis**
  - Scalable & accurate (context-sensitive, flow-sensitive) alias analysis is needed
  - Otherwise, model is either too big or too inaccurate
- **Long loops / large arrays**
  - Most frequent reason for not finding deep bugs in sequential programs
- **Heap shapes and properties**
  - Active area of research, but methods do not scale beyond a few KLOC yet
  - Can be leveraged to provide sound reductions for other properties
- **Modular verification and component interfaces**
  - Orthogonal to component-based methods, can provide scaling up to systems
  - Practical difficulties can be addressed by systematic development practices, but there should be a clear return on invested effort
- **Concurrent program verification**
  - Concurrency is pervasive, and very difficult to verify
  - Blowup in interleaved executions (in addition to issues in sequential programs)
  - Existing methods have limitations
    - Static program analysis: too many false warnings
    - Model checking: does not scale
    - Testing: poor coverage
  - **Great opportunity to contribute, especially with the proliferation of multi-cores!**

Thank you!

20th International Conference on
Computer Aided Verification
CAV 2008
July 7 – 13, 2008
Princeton, USA

http://www.princeton.edu/cav2008