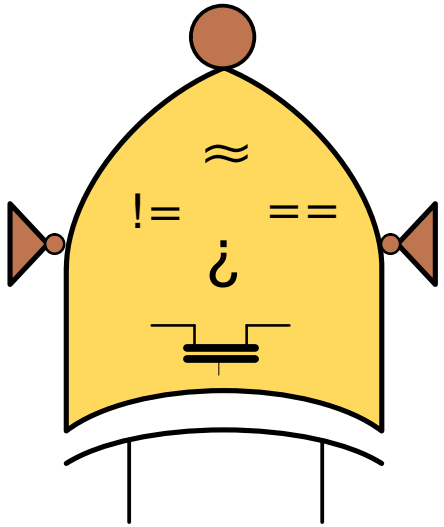
Functional Equivalence Checking in Practice



Andreas Kuehlmann
IBM T.J. Watson Research Center
Yorktown Heights, NY

contributions by
Florian Krohm, IBM Research Yorktown
Mark Williams, Jessie Xu, Arjen Mets, IBM EDA Fishkill

Three Generations of Equivalence Checking in IBM

1. SAS (1978 – 1994):

– standard equivalence checking tool based on mainframe
– based on DBA algorithm (BDDs in time)
– verified manual (later synthesized) cell–based designs against RTL spec
– handling of complex designs:
. application of proper cutpoints (compose does not exist in DBA)
. application of special synthesis routines to make designs similar
. special hacks for each hard problem

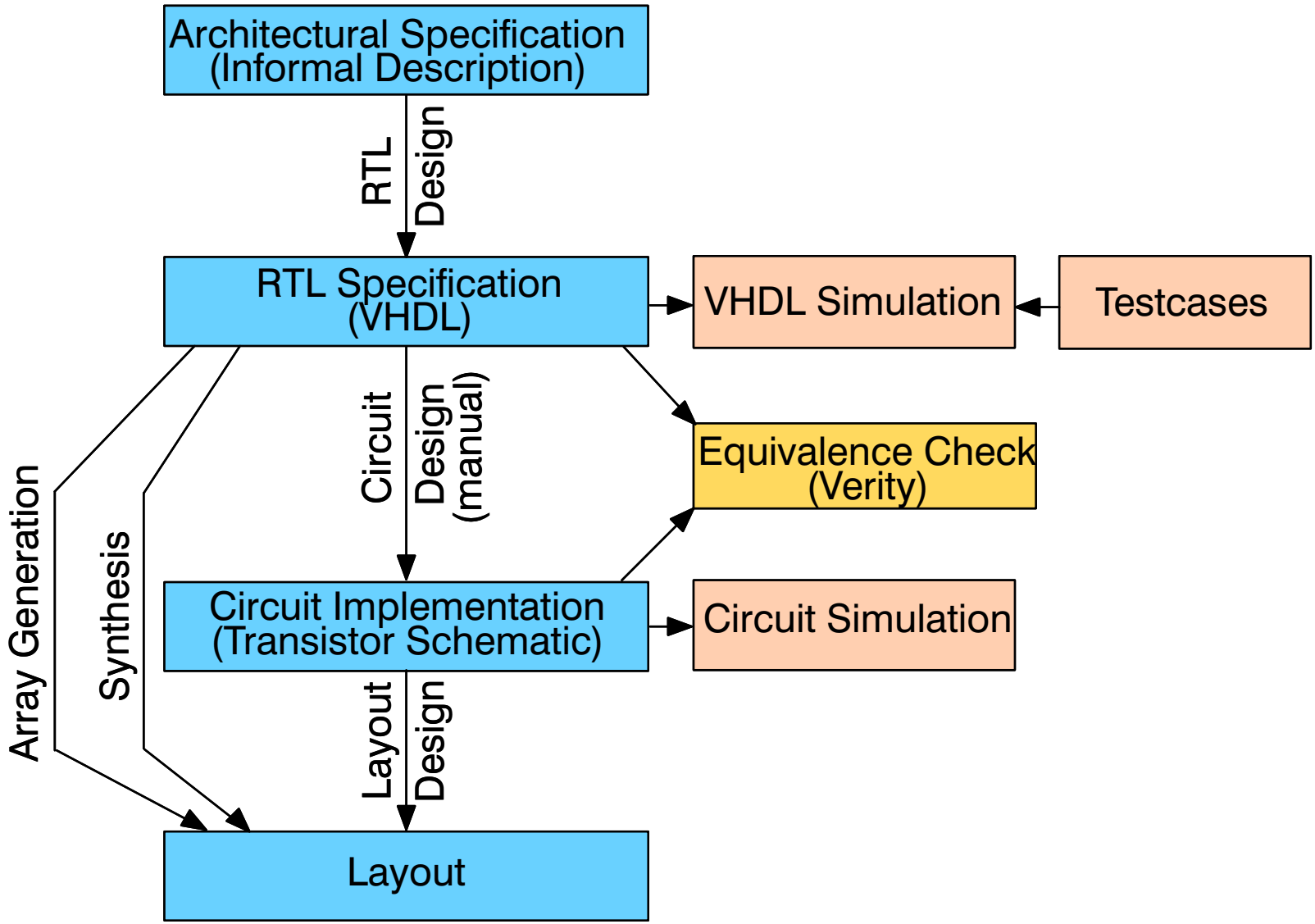
2. BEC (1991 – today):

– workstation based equivalence checker based on DBA
– mainly used in synthesis environment

3. Verity (1992 – today):

– equivalence checker for RTL, gate–level, and switch–level
– based on BDDs, random simulation and ATPG
– initially used to verify custom designed microprocessors
– today used for all combinations of switch–level, gate–level, and RTL designs
– becomes standard equivalence checker within IBM

Application of Equivalence Checking in μP Design



Verity

Verification Model:

- combinational verification model
- automatic register correspondence
- support of hierarchical and incremental verification
- programmable mixed–mode extraction scheme
  for wide variety of circuit techniques
  (static, ratioed logic, n–phase dynamic, ripple domino CMOS, etc.)

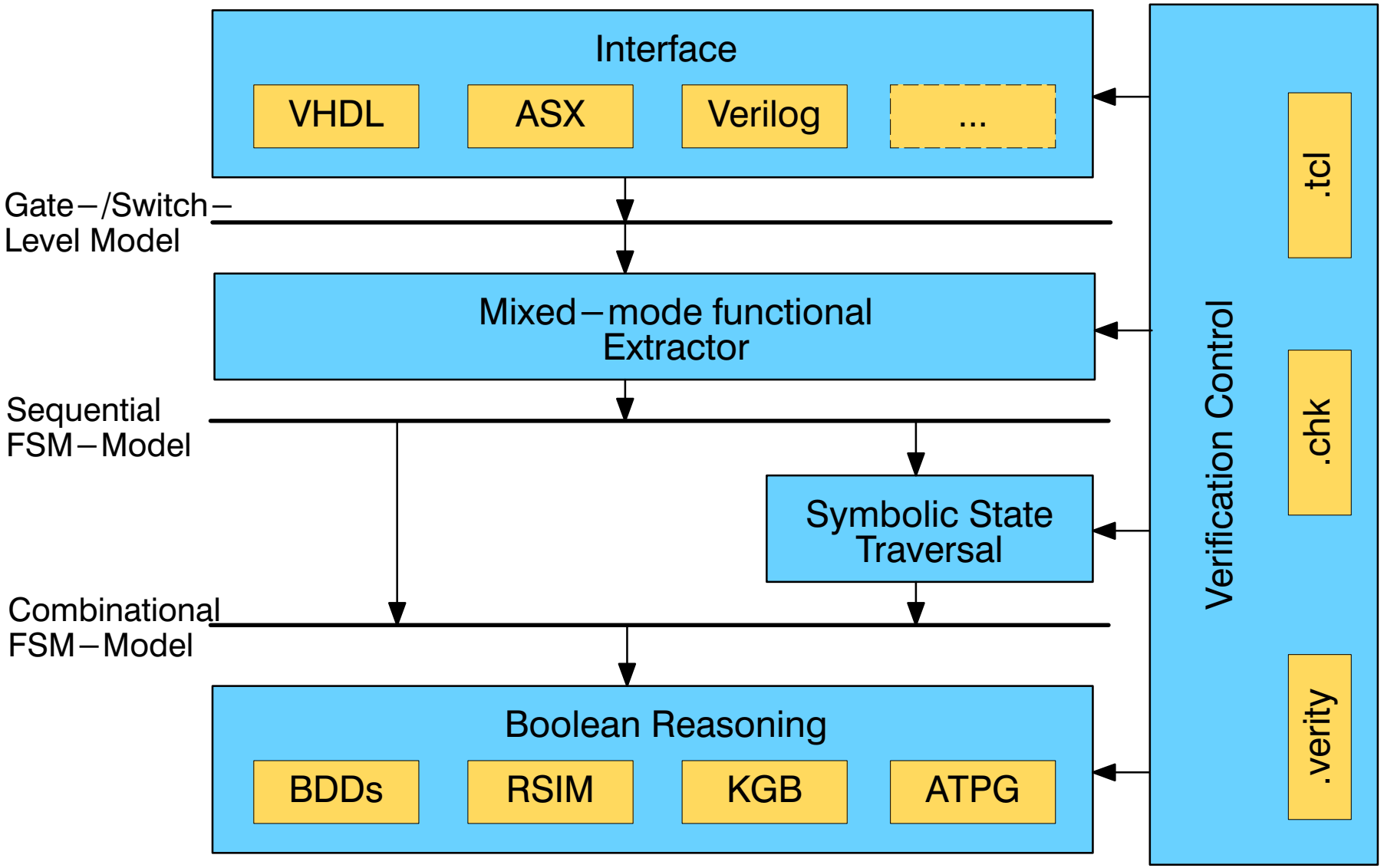
Verification Engines:

- random simulation
  - . effective for miscompares
  - . exhaustive simulation of small cones
- BDDs
  - . general and fast
  - . lot of code or pre–ordering, dynamic variable ordering, and assertion handling
  - . slow and memory expensive when BDDs get large
  - . ineffective if two sides are somewhat structurally similar
- ATPG
  - . based on permissible functions):
  - . effective for synthesizes components which
  - . ineffective if no name correspondence can be exploited

Implementation:

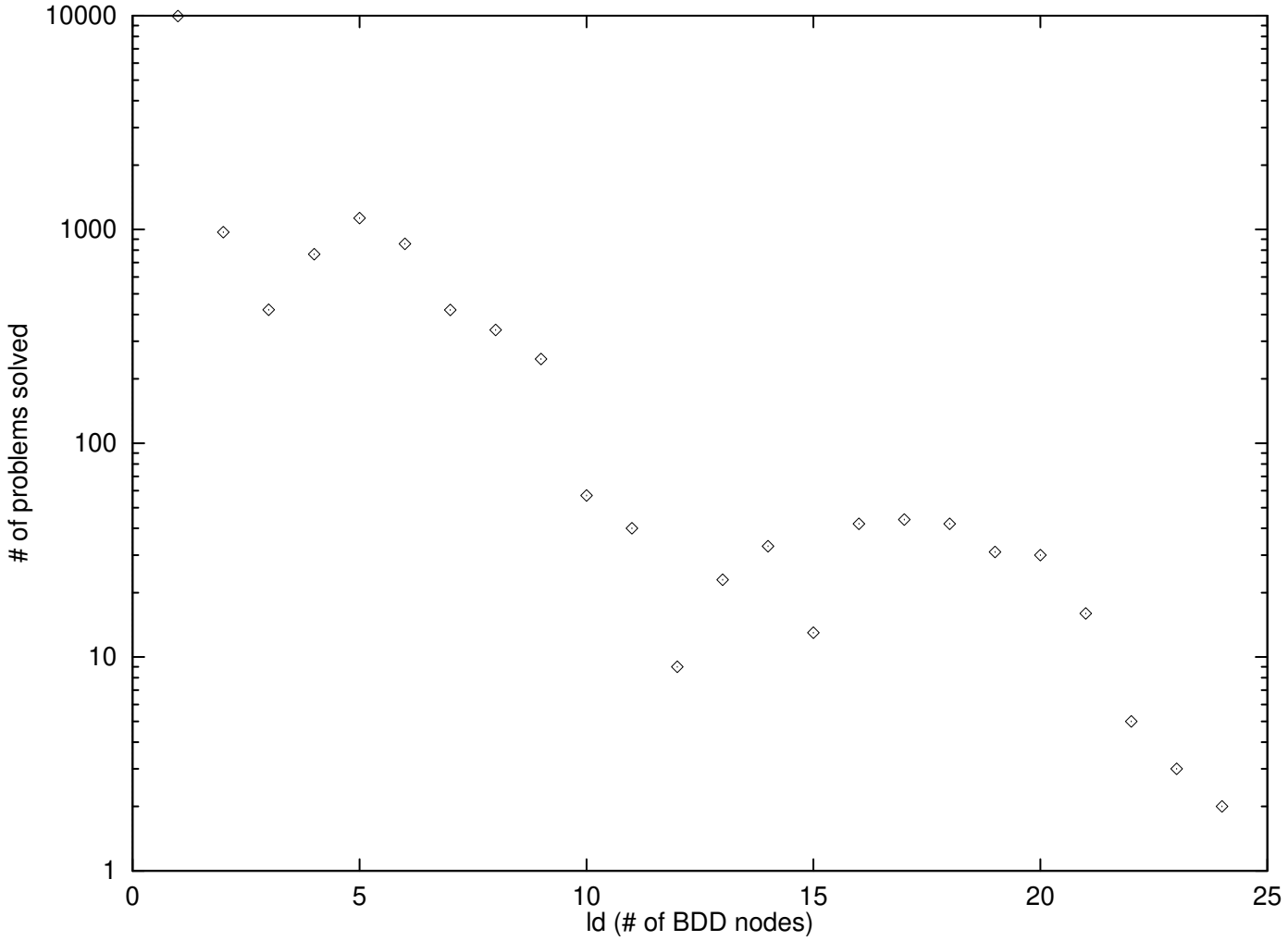
- interactive and batch mode, integration into schematic entry tools

Software Structure of Verity



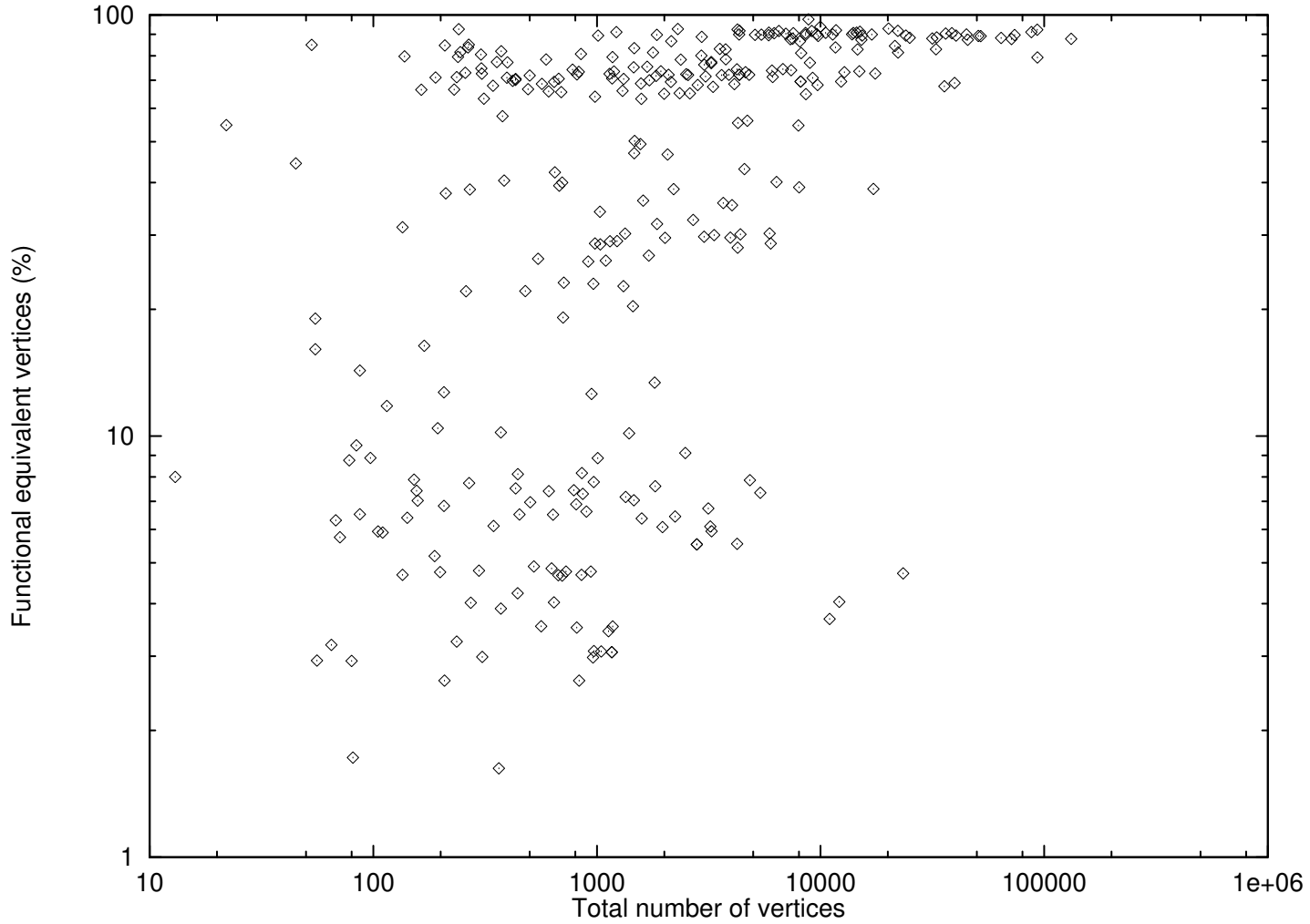
Do we really need huge BDDs?

How many comparisons require a certain BDD size:

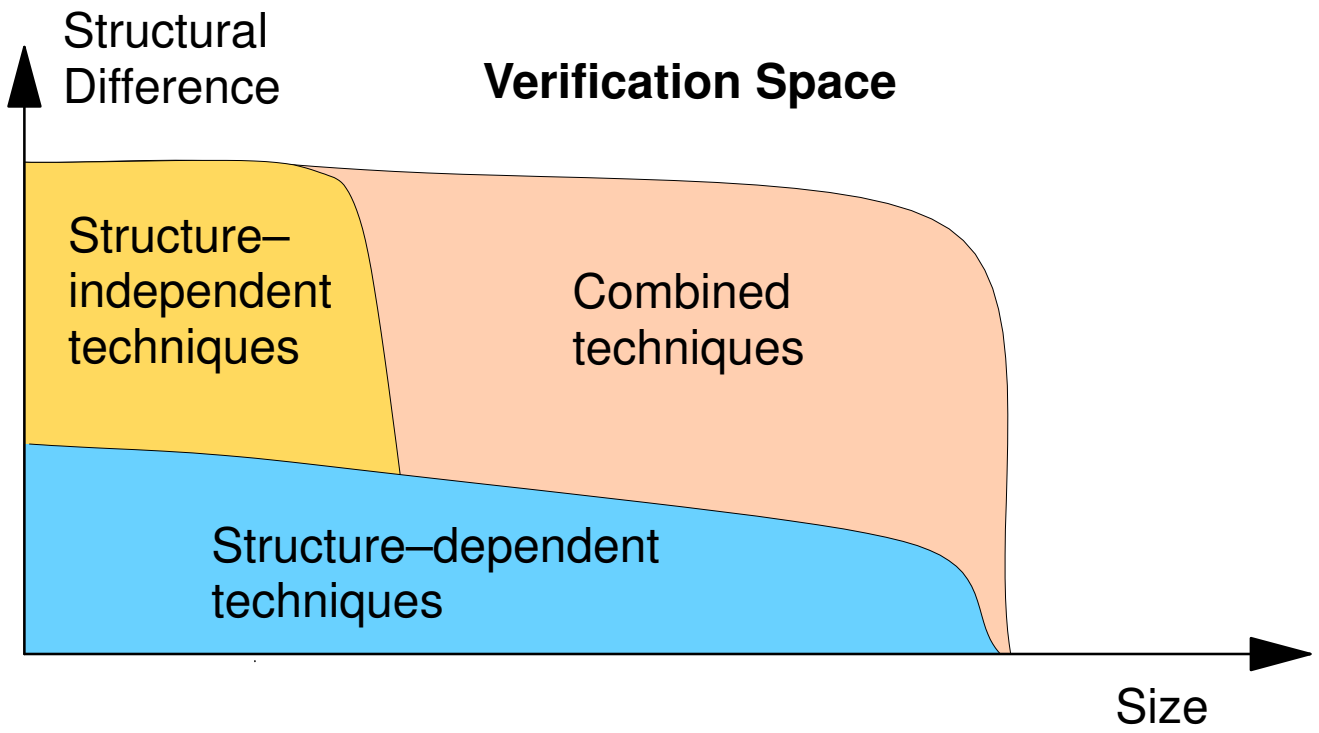


How similar are circuits in reality?

Number of functionally equivalent nets versus total number of nets in typical circuits:



Classification of Verification Approaches



Structure–independent techniques:

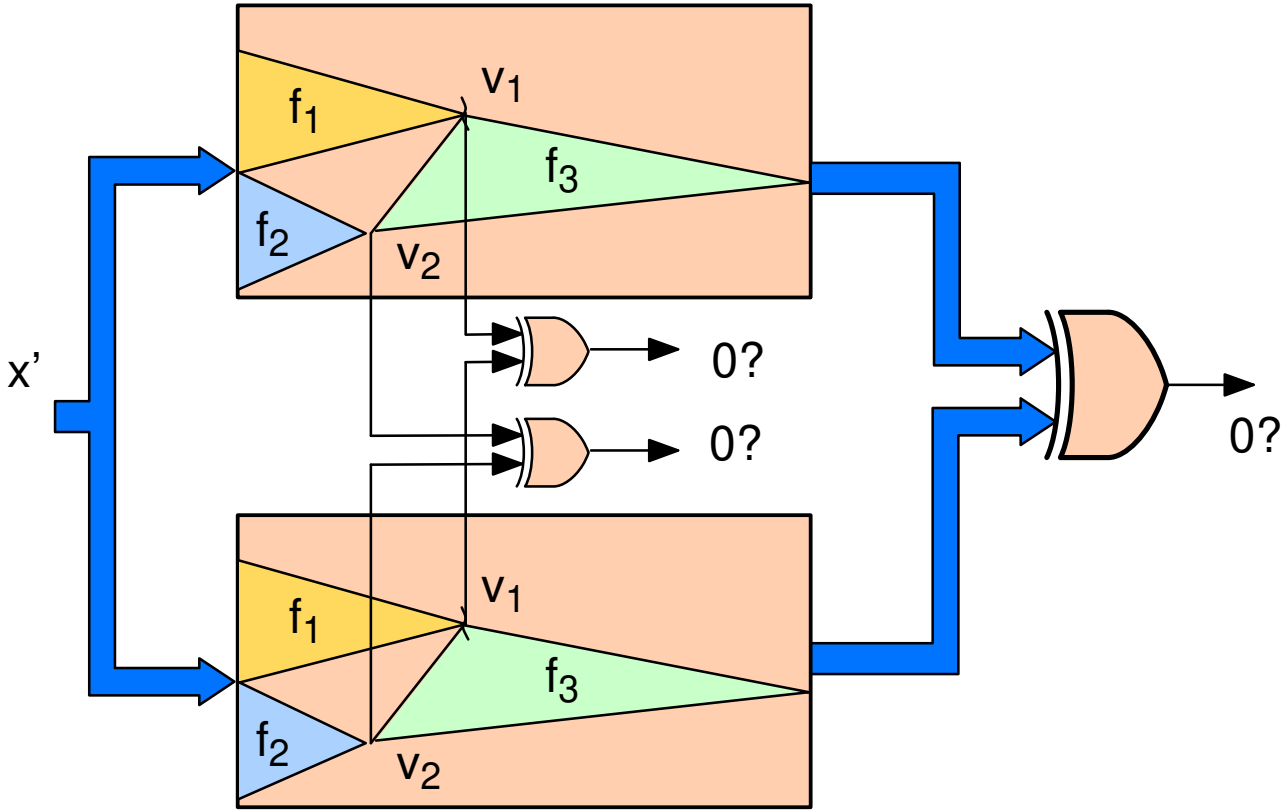
- exhaustive simulation,
- decision diagrams (*DD)
- functional hashing

Structure–dependent techniques:

- graph comparison
- test–pattern generators
- recursive learning
- decision diagrams combined with design partitioning (cut–points

Conventional Cut–Point Guessing

Verification of both circuits is partitioned by ”cut–points” into smaller portions.



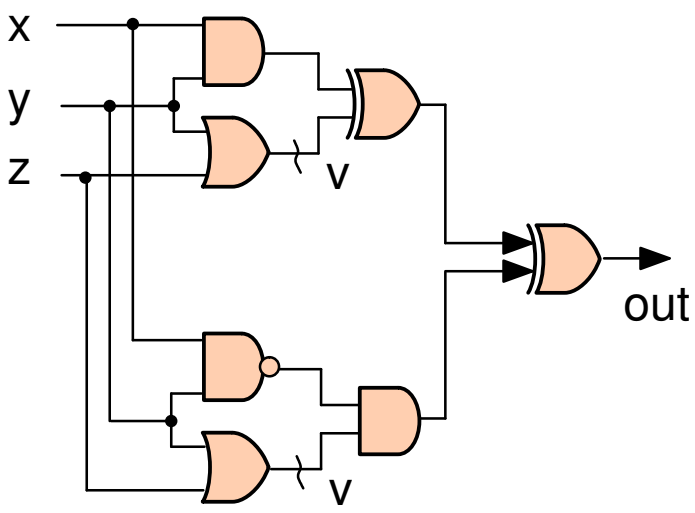
Cut–point guessing with BDD’s and random simulator:

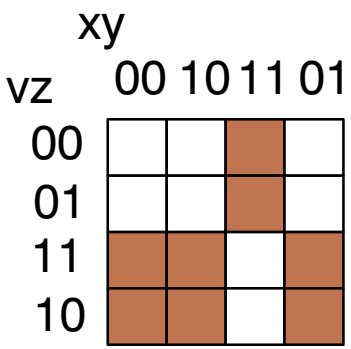
1. compute net signatures with random simulator
2. sort signatures + choose cut–points
3. iteratively verify and refine cut–points
4. verify outputs

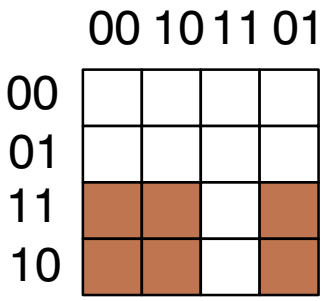
False Negatives

Outputs might **miscompare** for invalid assignment of cutpoint values !

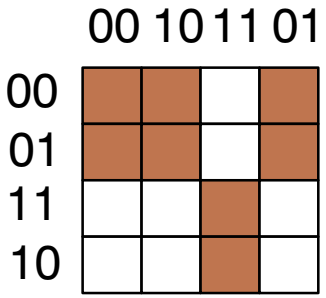
Example:







Valid setting of v:
$c = (v \equiv y \vee z)$

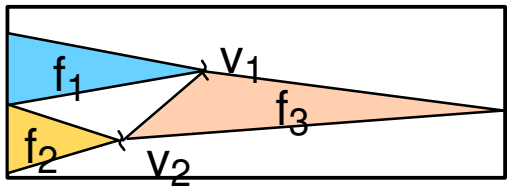


Eliminating of False Negatives

1. Resubstitution of cut variables by driving functions (`bdd_compose`)

Problem:
- order in which resubstitution is done is very critical
- "good" cutpoint might look bad in the presence of other bad cutpoints
  ⇒ results in blow–up of BDDs if resubstitued first
- ordering problem becomes worse with increasing # of cutpoints
  ⇒ selection of right cutpoints is crucial

2. Applying the image of previous cut frontiers to constrain the miscomparing function



constraint: $c = (v_1 \equiv f_1) \wedge (v_2 \equiv f_2)$

Problem:
- BDD of image blows up in practice (similar to sequential verification)

3. Disprove each counter example using ATPG

Problem:
- number of counter examples might be exponentially large
- proof might take exponential amount of time

Ingredients to KGB

– Hashing of the circuit graph:

. circuit is modeled as graph using ANDs and inverted edges only

. computed table (hash table) is used to discover structurally equivalent vertices

– Special BDD propagation scheme:

. BDD construction is **not done** in traditional DFS or BFS manner

Instead: Controlled by a heap sorted by the BDD size

Effect: Small BDD get processed first

. processing of BDDs is limited to a maximum size and discontinues if limit is exceeded

. vertices with equivalent BDDs are merged immediately

⇒ fanout structure is immediately rehashed

– Propagation of multiple BDD layers:

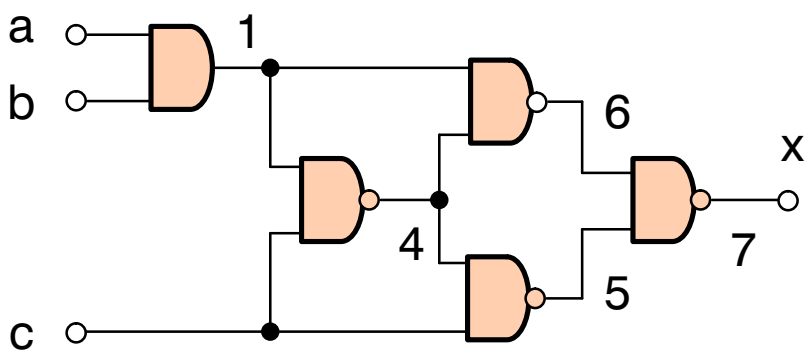
. insertion of cut frontiers based on all merged vertices

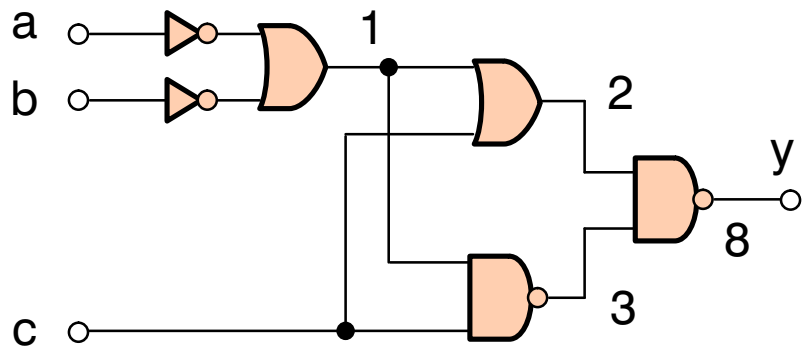
. overlapped propagation of multiple BDD layers starting at the cut frontiers

. in case of false negatives: heap controlled resubstitution of cut variables

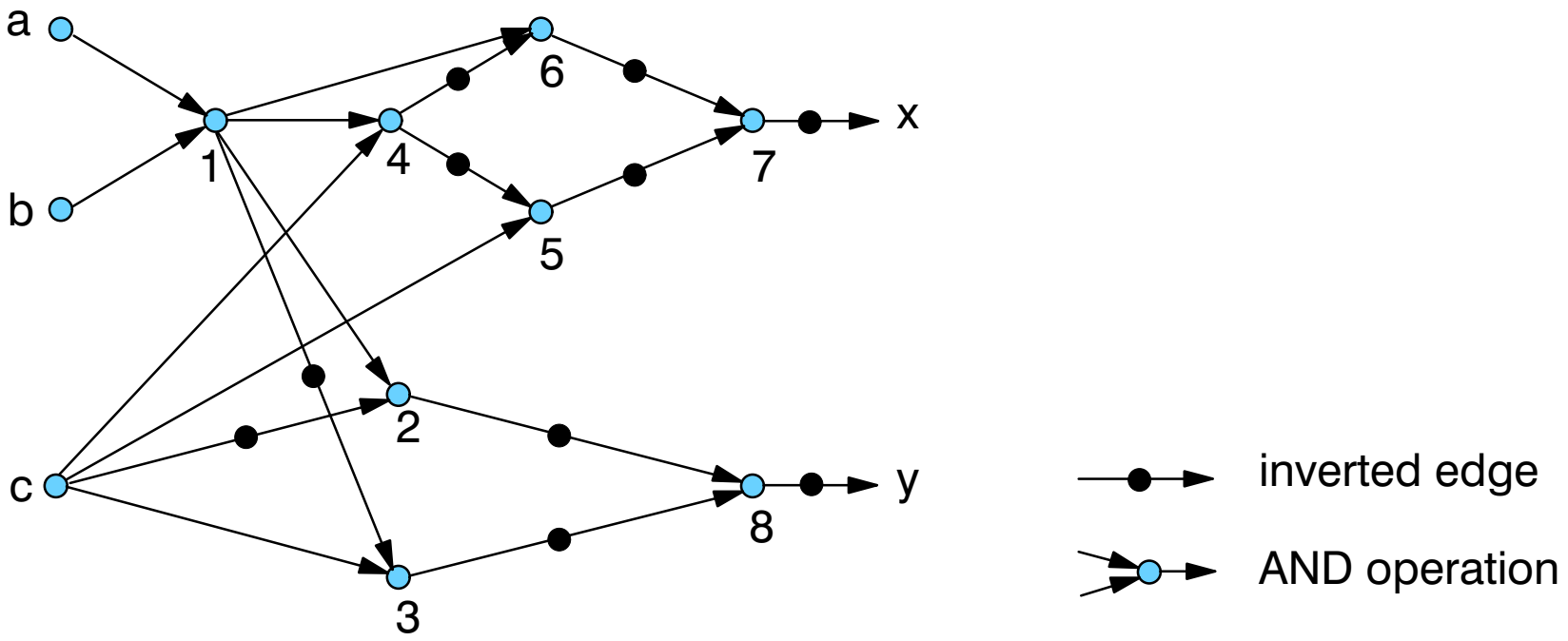
⇒ avoids blow–up of `bdd_compose`

Circuit Graph Hashing





Initial circuit graph:



Heap–based BDD Propagation

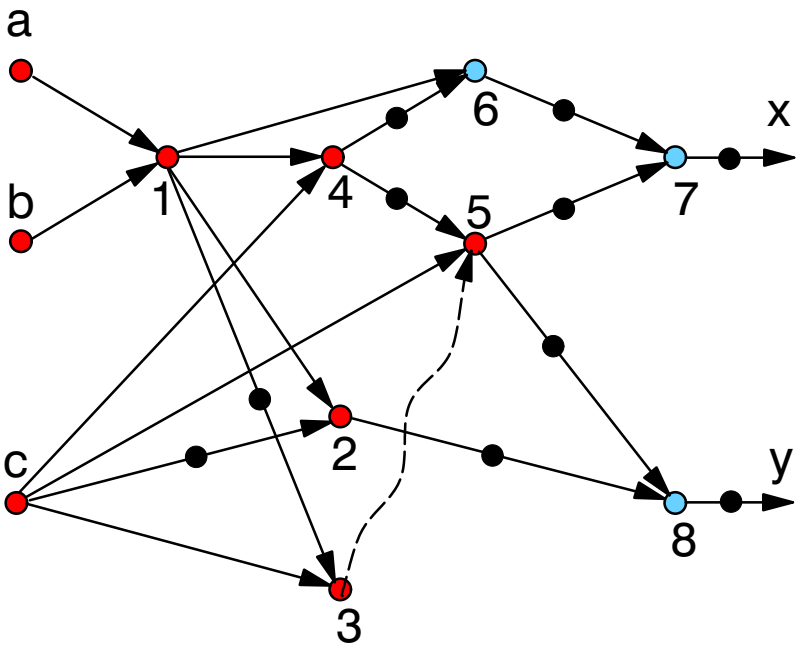
```
Algorithm Check_Equivalence (v1,v2) {
  if (v1 == v2)       return equal;       /* structural equiv. */
  if (v1 == NOT(v2))  return not_equal;

  for all primary inputs i do {           /* inject input BDDs */
    bddi = create_bdd_variable();
    store_vertex_at_bdd(bddi,i);
    put_on_heap(heap,bddi);
  }
  return process_heap(heap,v1,v2);
}
```

Heap Processing

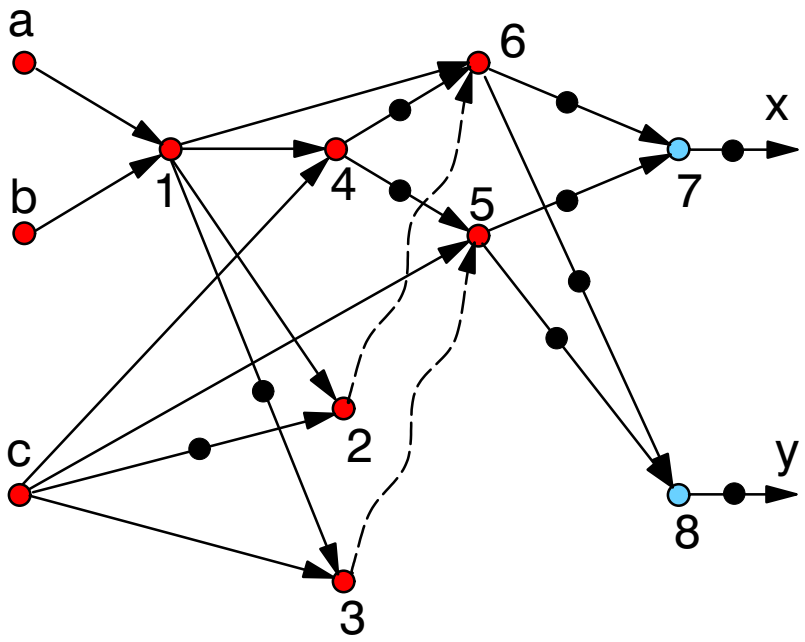
```
Algorithm process_heap(heap,v1,v2) {
  while (heap != empty) {
    bdd  = get_smallest_bdd(heap);
    v    = get_vertex_from_bdd(bdd);
    store_bdd_at_vertex(v,bdd);
    for all fanout vertices vout of v do {
      bddleft  = get_bdd_from_vertex(vout->left);
      bddright = get_bdd_from_vertex(vout->right);
      bddres   = bdd_and(bddleft,bddright);
      vres     = get_vertex_from_bdd(bddres);
      if (vres) {
        merge_vertices(vres,vout);  /* including rehashing */
        if (v1 == v2)          return equal;
        if (v1 == NOT(v2))     return not_equal;
      }
      else {
        store_vertex_at_bdd(bddres,vout);
      }
    }
    put_on_heap(heap,bddres);
  }
  return undecided;
}
```

BDD Propagation

Graph after building BDDs for
1,2,3,4,5 ⇒ 5 and 3 get merged



Graph after building BDDs for
6 ⇒ 6 and 2 get merged

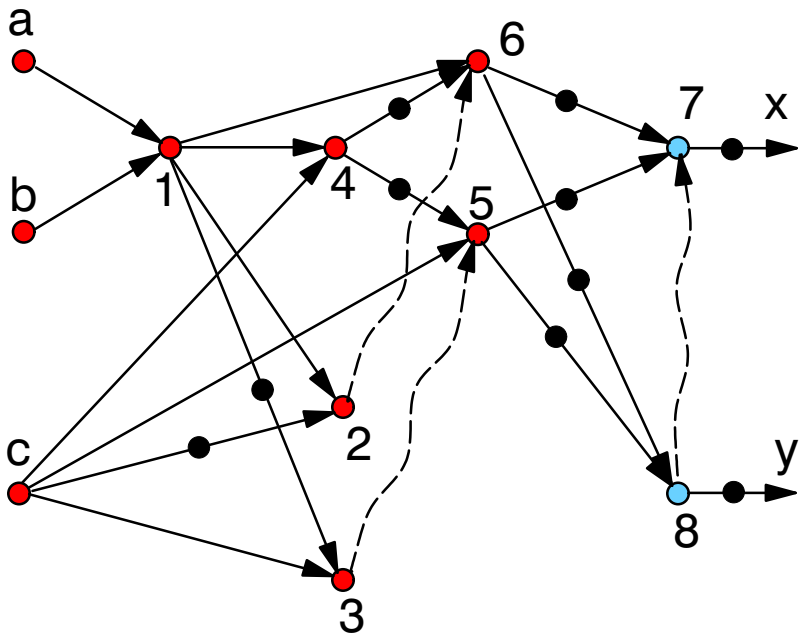


BDD built

no BDD built

Rehashing of Fanout Structure

Forward hashing merges 7 and 8

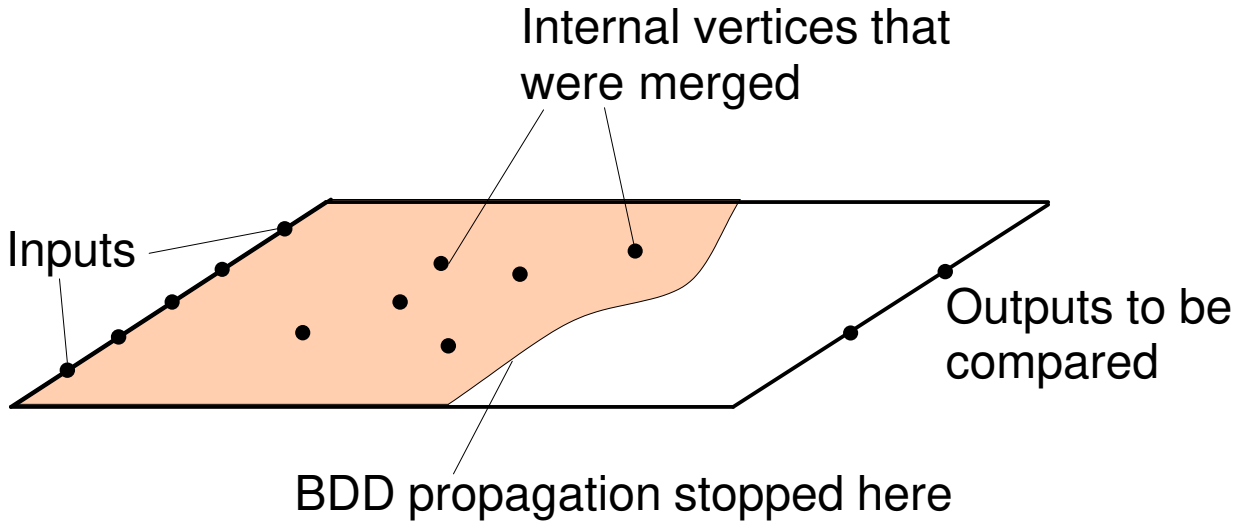


BDD built

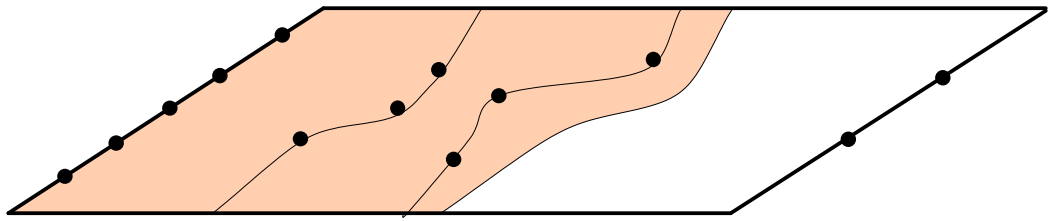
no BDD built

Cut Point Insertion

After BDD propagation

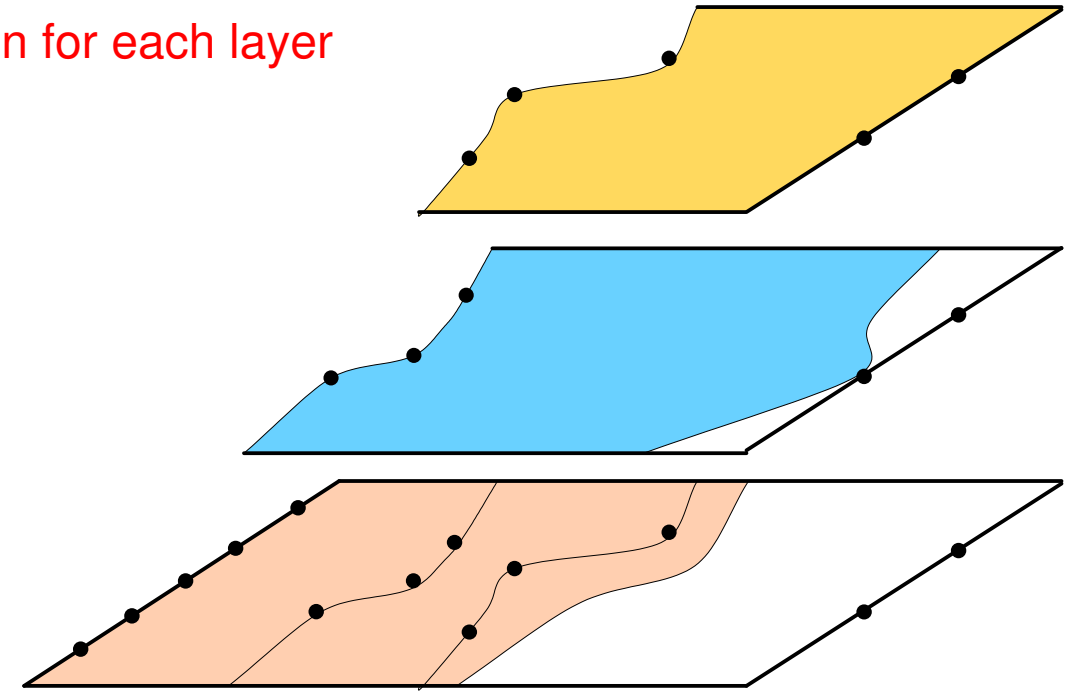


Insertion of cut frontiers based on topological order



BDD Propagation from Cut Frontiers

Independent BDD propagation for each layer



Algorithms for heap processing is modified:
- add level information to BDD node
- when new BDD is generated, assign to maximum level of both input BDDs

Heap–based False Negative Elimination

```
Algorithm Eliminate_False_Negatives (v1,v2) {
  for level = 0 to c_level(v1) do {
    bdd = get_bdd_from_vertex(v1,level);
    put_on_heap(compose_heap,bdd);
  }
  for level = 0 to c_level(v2) do {
    bdd = get_bdd_from_vertex(v2,level);
    put_on_heap(compose_heap,bdd);
  }
  return process_compose_heap(heap,v1,v2);
}
```

Compose Heap Processing

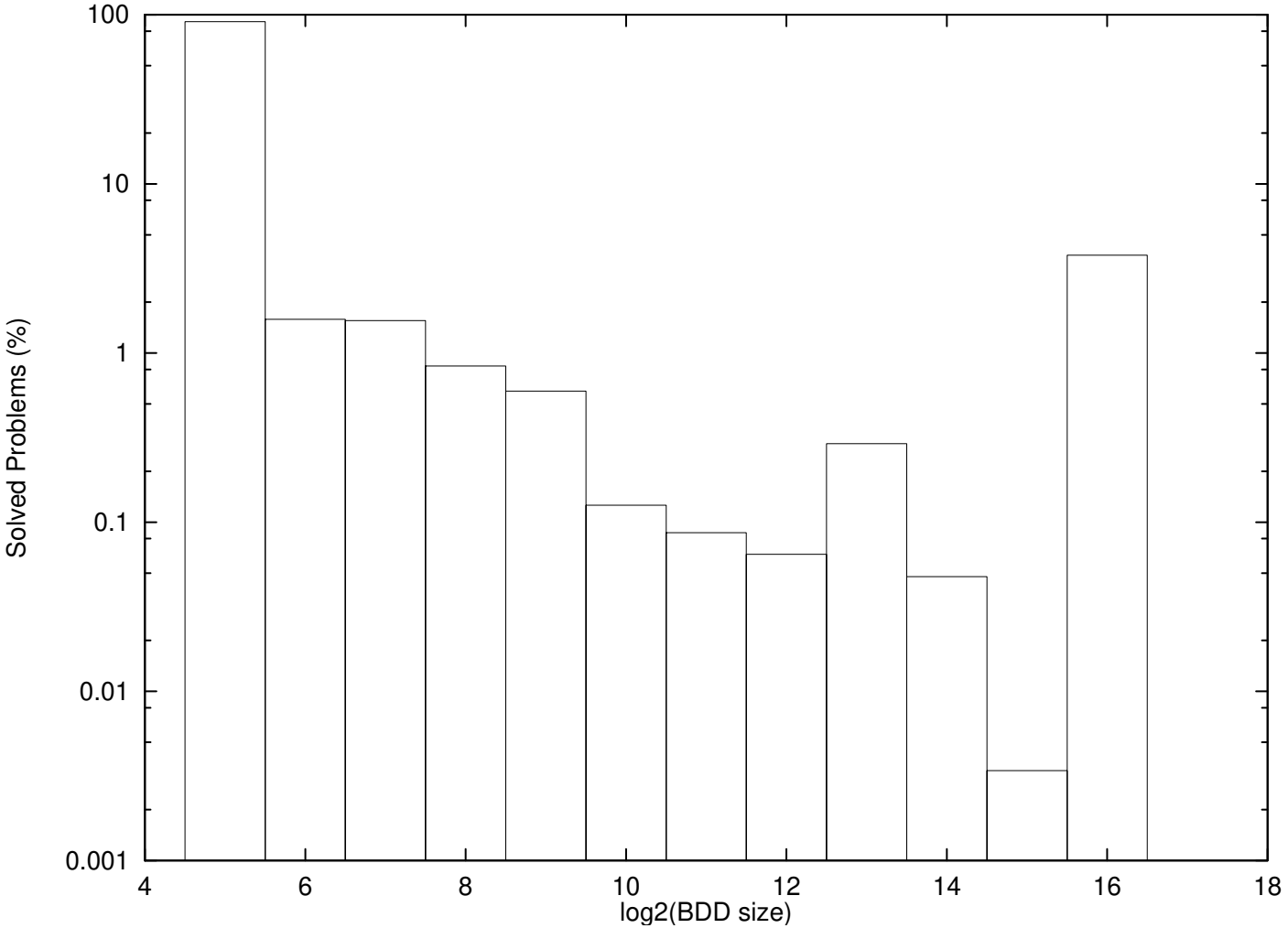
```
Algorithm process_compose_heap(heap,v1,v2) {
  while (compose_heap != empty) {
    bdd        = get_smallest_bdd(compose_heap);
    v          = get_vertex_from_bdd(bdd);
    bdd_var    = get_cutcar_from_bdd(bdd);
    if (bdd_var) {
      v_var    = get_vertex_from_bdd(bdd_var);
      level    = c_level(v_var);
      bdd_func = get_bdd_from_vertex(v_var,level-1);
      bdd_res  = bdd_compose(bdd,bdd_var,bdd_func);
      v_res    = get_vertex_from_bdd(bdd_res);
      if (v_res) {
        merge_vertices(v_res,v_out);
        if (v1 == v2)          return equal;
        if (v1 == NOT(v2))     return not_equal;
      }
      else {
        store_vertex_at_bdd(bdd_res,v_out);
        put_on_heap(compose_heap,bdd_res);
      }
    }
  }
  return undecided;
}
```

Results

Design	Inputs/ Outputs	Gates/ Transistors	Compars./ Checks	CPU (sec)	Memory (MB)
C1355	4 / 32	1940 / –	64 / 4008	24.3	2.9
C1908	33 / 25	2536 / –	50 / 5730	6.2	2.9
C2670	233 / 139	2637 / –	278 / 6324	3.3	4.0
C3540	50 / 22	4113 / –	44 / 9140	29.21	7.4
C432	36 / 7	618 / –	14 / 1236	11.3	8.5
C499	41 / 32	484 / –	64 / 968	16.7	2.4
C5315	178 / 123	6213 / –	246 / 13714	46.3	15.7
C6288	32 / 32	9036 / –	64 / 18072	9.6	6.7
C7552	207 / 108	8916 / –	216 / 20050	14.5	6.7
µP w/ BB	202615 / 84354	– / –	406068 / 3109078	13922.4	670.3
µP flat	47370 / 5802	1024763 / 14460	93236 / 687792	1785.7	552.7
MULT	2383 / 3325	28432 / 57806	7316 / 86066	3437.7	83.1
DIFF1	1519 / 482	29807 / –	3108 / 44765	1760.9	31.8
DIFF2	11661 / 2398	51759 / –	23838 / 100204	136.2	25.1

How big are the BDDs now?

Number of verification problems solved with a given BDD size limit:



Conclusion

GREAT