A Single-Instance Incremental SAT Formulation of Proof- and Counterexample-Based Abstraction

Niklas Een, Alan Mishchenko
EECS Department
Univ. of California, Berkeley, USA.

Nina Amla
Cadence Research Labs
Berkeley, USA.

Abstract. This paper presents an efficient, combined formulation of two widely used abstraction methods for bit-level verification: *counterexample-based abstraction (CBA)* and *proof-based abstraction (PBA)*. Unlike previous work, this new method is formulated as a single, incremental SAT-problem, interleaving CBA and PBA to develop the abstraction in a bottom-up fashion. It is argued that the new method is simpler conceptually and implementation-wise than previous approaches. As an added bonus, proof-logging is not required for the PBA part, which allows for a wider set of SAT-solvers to be used.

1 Introduction

Abstraction techniques have long been a crucial part of successful verification flows. Indeed, the success of SAT-solving can largely be attributed to its inherent ability to perform localization abstraction as part of its operations. For this reason so called bug-hunting, or BMC, methods can often be applied on a full design directly, thereby deferring the abstraction work to the SAT-solver. However, computing an abstraction explicitly is often more useful for hard properties that require a mixture of different transformation and proof-engines to complete the verification.

In our formulation, both CBA and PBA compute a localization in the form of a set of flops. An abstracted flop is in essence replaced by a primary input (PI), thus giving more behaviors to the circuit. Both methods work by analyzing, through the use of SAT, a k-unrolling of the circuit. However, they differ as follows:

- CBA works in a bottom-up fashion, starting with an empty abstraction (all flops are replaced by PIs) and adding flops to refute the counterexamples as they are enumerated for successively larger k.
- PBA, in contrast, considers the full design and a complete refutation of all counterexamples of depth k (in the form of an UNSAT proof). Any flop not syntactically present in the proof of UNSAT is abstracted.

The two methods have complementary strengths: CBA by virtue of being bottom-up is very fast, but may include more flops than necessary. PBA on the other hand does a more thorough analysis and almost always gives a tighter abstraction than CBA, but at the cost of longer runtime.

In this work, it is shown how the two methods can be seamlessly combined by applying PBA, not on the full design, but on the latest abstraction produced by CBA. This solution has a very elegant incremental SAT formulation, which results in a simple, scalable algorithm that has the strength of both methods.

In the experimental section it is shown how a design with 40,000 flops and 860,000 AND-gates is localized to a handful of flops in just 4 seconds (much faster than any previous method), and how this abstraction is instantaneously solved by the interpolation-based proof-engine [10], whilst the original unabstracted design took 2 minutes to verify, despite the inherent localization ability of interpolation.

2 Related Work

Counterexample-based abstraction was first introduced by Kurshan in [8] and further developed by Clarke et. al. in [3]. Proof-based abstraction was coined by McMillan [11], and independently proposed by Gupta et. al in [7].

The work most closely related to ours is Gupta's work of [7] and McMillan et. al's work of [1]. In both approaches, abstract counterexamples are concretized using a SAT-solver. When concretization fails, the UNSAT proof guides the abstraction refinement. Our work does not rely on a SAT-solver to refute counterexamples, but instead uses a simpler and more scalable method based on ternary simulation (section 4.1).

Gupta's approach does not rely on BDD reachability to produce abstractions; although BDDs are used to form a complete proof-procedure. Like our method, it tries to limit the amount of logic that is put into the SAT-solver when unrolling the circuit, thereby improving scalability. It differs, though, in that the initial unrolling is done on the concrete design (our method starts with an empty abstraction), and that PBA is used to shrink the abstract model in a more conservative manner, requiring the PBA result to stabilize over several iterations.

The work of McMillan et. al. mixes PBA for refuting all counterexamples of length k with proof-analysis of counterexamples from the BDD engine, refuting individual (or small sets of) counterexamples. Unlike Gupta's work, BDDs are an integral part of the abstraction computation.

The approach proposed in this paper differs further from

previous work in that it does not constitute a complete proof-procedure. There are many different ways of using an abstraction method as part of a verification flow. A simple use-model would be: Run the abstraction computation until some resource limit is reached, then output the best abstraction found so far and put the method on hold. If the abstraction turns out not to be good enough for the downstream flow, resume abstraction computation with a higher resource limit, and produce a more refined abstraction. Obviously, this use-model can be further improved by multi-threading on a multi-core machine.

In the experimental evaluation, we choose to pass abstractions to an interpolation-based proof-engine. This particular setup relates to the work of [9] and [2].

3 Assumptions and Notation

In the presentation, the following is assumed:

- The design is given as a set of next-state functions expressed in terms of current state variables (flops) and primary inputs (PIs).
- The design has only one property, which is a safety property.
- All flops are initialized to zero, and are running on the same clock (hence acting as unit delays).

It is further assumed that the logic of the next-state functions is represented as a combined And-Inverter-graph, with the single property being the output of a particular AND-gate. As customary, the negation of the property is referred to as the *bad* signal.

An "abstraction" is identified with a set of flops. If a flop is not part of the abstraction, it is treated as a PI in the abstract model of the design. By this semantics, adding a flop to the current abstraction means *concretizing* it in the abstract model: replace the PI by a flop and connect it to the appropriate input signal.

4 Algorithm

How does the proposed algorithm work? It starts by assuming the empty abstraction, treating all flops as PIs. It then inserts one time-frame of the design into the SAT-solver, and asks for a satisfying assignment that produces TRUE at the *bad* signal. The SAT-solver will come back SAT[1] and the counterexample is used to concretize some of the flops (= CBA). When enough flops have been concretized, the SAT-problem becomes UNSAT, which means that all counterexamples of length 0 have been refuted (unless there is a true counterexample of length zero). The algorithm can now move on to depth 1, but before doing so, any flop that did not occur in the UNSAT proof is first removed from the abstraction (= PBA). The procedure is repeated for increasing depths, resulting in an incremental sequence of SAT calls that looks something like

depth 0: SAT, SAT, SAT, SAT, **UNSAT**
depth 1: SAT, **UNSAT**
depth 2: SAT, SAT, SAT, **UNSAT**
⋮

with each sequence of calls at a given depth ending in an "UNSAT" result that prunes the abstraction built up by analyzing the preceding "SAT" counterexamples.

The algorithm terminates in one of two ways: either (i) CBA comes back with the same set of flops as were given to it, which means we have found a true, justified counterexample, or (ii) it runs out of resources for doing abstraction and stops. The resulting abstraction is then returned to the caller to be used in the next step of the verification process.

4.1 Counterexample-based refinement

Assume that for the current abstraction A the last call to SAT returned a counterexample of length k. The counterexample is then analyzed and refined by the following simple procedure[2] in order to refute it:

> **CBA refinement.** Loop through all flops not in A. Replace the current value of the counterexample with an X (the undefined value) and do a three-valued simulation. If the X does not reach the *bad* signal, its value is unimportant for the justification of the counterexample, and the corresponding flop is kept as a PI. If, on the other hand, X propagates all the way to *bad*, we undo the changes made by that particular X-propagation and add the corresponding flop to A.

The order in which flops are inspected does matter for the end result. It seems like a good idea to consider multiple orders and pick the one producing the smallest abstraction. But in our experience it does not improve the overall algorithm. The extra runtime may save a few flops temporarily, but they are typically added back in a later iteration, or removed by PBA anyway, resulting in the same abstraction in the end.

4.2 Incremental SAT

Incremental SAT is not a uniquely defined concept. The interpretation used here is a solver with the following two methods:

- ***addClause***(*literals*): This method adds a clausal constraint, i.e. $(p_0 \vee p_1 \vee \ldots \vee p_{n-1})$ where $p_i \in$ *literals*, to the SAT-solver. The incremental interface allows for more clauses to be added later.
- ***solveSat***(*assumps*): This method searches for an assignment that satisfies the current set of clauses under the unit assumptions $assumps = a_0 \wedge a_1 \wedge \ldots \wedge a_{n-1}$. If there is an assignment that satisfies all the clauses added so far, as well as the unit literals a_i, that model

[1] The very first query only comes back "UNSAT" if the property holds combinationally, a corner case we ignore here.

[2] This procedure (implemented by Alan Mishchenko in ABC [6]), has been independently discovered by one of our industrial collaborators, and probably by others too. A similar procedure is described in [13].

is returned. If, on the other hand, the problem is UNSAT under the given assumptions, *the subset of those assumptions used in the proof of UNSAT* is returned in the form of a final conflict clause.

The extension of ***solveSat***() to accept a set of unit literals as assumptions, and to produce the subset of those that were part of the UNSAT proof, can easily be added to any modern SAT-solver.[3] This is in contrast to adding proof-logging, which is a non-trivial endeavor. For that reason, the proposed algorithm is stated entirely in terms of this interface and does not rely on generating UNSAT proofs.

4.3 Refinement using activation literals

Unlike the typical implementation of PBA, this work uses *activation literals*, rather than a syntactic analysis of resolution proofs, to determine the set of flops used for proving UNSAT. For each flop f that is concretized, a literal a is introduced in the SAT-instance. As the flop input f_{in} at time-frame k is tied to the flop output at time-frame $k+1$, the literal is used to activate or deactivate propagation through the flop by inserting two clauses stating:

$$a \rightarrow (f[k+1] \leftrightarrow f_{in}[k])$$

The set of activation literals is passed as assumptions to ***solveSat***(), and for UNSAT results, the current abstraction can immediately be pruned of flops missing from the final conflict clause returned by the solver.

This PBA phase is very affordable. The same SAT-problem would have to be solved in a pure CBA based method anyway. The cost we pay is only that of propagating the assumption literals. Because abstractions are derived in a bottom-up fashion, with the final abstraction typically containing just a few hundred flops, the overhead is small.

5 Implementation

This section describes the combined abstraction method in enough detail for the reader to easily and accurately reproduce the experimental results of the final section. The pseudo-code uses the following conventions:

- Symbol **&** indicates pass-by-reference.
- The type **Vec**⟨**T**⟩ is a dynamic vector whose elements are of type T.
- The type **Netlist** is an extended And-Inverter-graph. It has the following gate types: AND, PI, FLOP, CONST. Inverters are represented as complemented edges. Flops act as unit delays. Every netlist N, has a special gate $N.True$ of type CONST.
- The type **Wire** represents an edge in the netlist. Think of it as a pointer to a gate plus a "sign" bit. It serves the same function as a *literal* w.r.t. a variable in SAT. Function ***sign***(w) will return TRUE if the edge is complemented, FALSE otherwise. By w_0 and w_1 we refer to the left and right child of an AND-gate. By w_{in} we refer to the input of a flop.
- The type **WSet** is a set of wires.
- The type **WMap**⟨**T**⟩ maps wires to elements of type T. For practical reasons, the sign bit of the wire is *not* used. For map m, $m[w]$ is equivalent to $m[\neg w]$. Unmapped elements of m are assumed to go to a distinct element T_UNDEF (e.g. LIT_UNDEF for literals, or WIRE_UNDEF for wires).
- The type **lbool** is a three-valued boolean that is either *true*, *false*, or *undefined*, represented in the code by: LBOOL_0, LBOOL_1, LBOOL_X.
- Every SAT-instance S (of type **SatSolver**) has a special literal $S.True$ which is bound to *true*. Method S.***newLit***() creates a new variable and returns it as a literal with positive polarity. Clauses are added by S.***addClause***() and method S.***satSolve***() commences the search for a satisfying assignment.

Because the pseudo-code deals with two netlists N and F, wire-types are subscripted Wire_N and Wire_F to make clear which netlist the wire belongs to. The same holds for WSet and WMap.

5.1 BMC Traces

To succinctly express the SAT analysis of the unrolled design, the class **Trace** is introduced (see Figure 1). It allows for incrementally extending the abstraction, as well as lengthening the unrolled trace. Its machinery needs the following:

- A reference N to the input design (read-only).
- A set of flops *abstr*, storing the current abstraction. Calling ***extendAbs***() will grow this set. Calling ***solve***() may shrink it through its built-in PBA.
- A netlist F to store the unrolling of N under the current abstraction. Gates are put into F by calling ***insert***(*frame*, w). Only the logic reachable from gate w of time-frame *frame* is inserted. For efficiency, netlist F is kept structurally hashed.
- A SAT-instance S to analyze the logic of F. Calling ***solve***(*f_disj*) will incrementally add the necessary clauses to model the logic of F reachable from the set of wires *f_disj*. The user of the class does not have to worry about how clauses are added; hence ***clausify***() is a private method. The SAT-solving will take place under the assumption $f_0 \vee f_1 \vee \ldots \vee f_{n-1}$. The method ***solve***() has two important side-effects:

[3] Two simple things should be done: (i) the decision heuristic has to be changed so that the first n decisions are made on the assumption literals; and (ii) if a conflict clause is derived that contradicts the set of assumptions, that clause has to be further analyzed back to the decision literals rather than the first UIP. For more details, please review the ***analyzeFinal()*** method of MiniSAT [5].

```
class Trace {
 – Private variables:
      Netlist&                  N;
      Netlist                   F;
      SatSolver                 S;
      WSet_N                    abstr;      – publicly read-only

      Vec⟨WMap_N⟨Wire_F⟩⟩        n2f;
      WMap_F⟨Lit⟩               f2s;
      WMap_N⟨Lit⟩               act_lits;
 – Private functions:
      Lit  clausify    (Wire_F f);
      void insertFlop (int frame, Wire_N w_flop, Wire_F f);
 – Constructor:
      Trace(Netlist& N);
 – Public functions:
      Wire_F insert    (int frame, Wire_N w);
      void   extendAbs (Wire_N w_flop);
      bool   solve     (WSet_F f_disj);
      Cex    getCex    (int depth);
};

class Cex { ... };      – stores a counter-example
```

Figure 1. *Interface of the "Trace" class.* The class handles the BMC unrolling of the design *N*. Netlist *F* will store the structurally hashed unrolling of *N*. SAT-solver *S* will store a CNF representation of the logic in *F*.

  - For satisfiable runs, the satisfying assignment is stored so that ***getCex***() can later retrieve it.
  - For unsatisfiable runs, the flops not participating in the proof are *removed* from the current abstraction.

– Maps *n2f* and *f2s*. Expression "*n2f*[*d*][*w*]" gives the wire in *F* corresponding to gate *w* of *N* in frame *d*. Expression "*f2s*[*f*]" gives the literal in *S* corresponding to gate *f* of *F*.

– Map *act_lits*. Expression "*act_lits*[*w_flop*] gives the activation literal for flop *w_flop*, or WIRE_UNDEF if none has been introduced.

5.2 The main procedure

The main loop of the abstraction procedure is given in Figure 2. Trace instance *T* is created with an empty abstraction. For increasing depths, the following is done:

– If the SAT-solver produces a counterexample, it is analyzed (by ***refineAbstraction***()) and flops are added to the abstraction to rule out this particular counterexample.

– If UNSAT is returned, the depth is increased. The ***solve***() method will have performed proof-based abstraction internally and may have removed some flops from the abstraction.

For each new depth explored, a new *bad* signal is added to *bad_disj*. This disjunction is passed as an assumption to the *solve* method of *T*, which means we are looking for a counterexample where the property fail in at least one time frame. It is not enough to just check the last time frame because of PBA.

5.3 Unrolling and SAT solving

Figure 3 details how ***insert***() produces an unrolling of *N* inside *F*, and Figure 4 describes how ***solve***() translates the logic of *F* into clauses and calls SAT. Great care is taken to describe accurately what is implemented, as the precise incremental SAT formulation is important for the performance and quality. For the casual reader who may not want to delve into details, the following paragraph summarizes some properties of the implementation:

As the procedure works its way up to greater and greater depth, only the logic reachable from the *bad* signal is introduced into the SAT-solver, and only flops that have been concretized bring in logic from the preceding time-frames. Constant propagation and structural hashing is performed on the design, although constants are not propagated across time-frames due to proof-based abstraction (PBA). Concrete flops are guarded by activation literals, which are used to implement PBA. One literal guards all occurrences of one flop in the unrolling. Flops that are removed by PBA will not be unrolled in future time-frames. However, fanin-logic from removed flops will remain in F and in the SAT-solver, but is disabled using the same activation literals.

6 Evaluation and Conclusions

The method of this paper was evaluated along two dimensions: (i) how does the new abstraction procedure fare in the simplest possible verification flow, where a complete proof-engine (in this case interpolation [10]) is applied to its result versus applying the same proof-engine without any abstraction; and (ii) how does it compare to previous hybrid abstraction methods—in our experiments, the implementation of CBA and PBA inside ABC [6], and the hybrid method of McMillan et. al. [1].

The examples used were drawn from a large set of commercial benchmarks by focusing on designs with local properties containing more than 1000 flops.[4] Experiments were run on an 2 GHz AMD Opteron, with a timeout of 500 seconds. The results are presented in Table 1.

For all methods, the depth was increased until an abstraction good enough to prove the property was found. ABC has a similar CBA implementation to the one presented in this work (based on ternary simulation), but restarts the SAT-solver after each refinement. ABC's PBA procedure is separate from CBA, so we opted for applying it once at the end to trim the model returned by CBA. This flow was also

[4] In other words, we've picked examples for which abstraction should work well. There are many verification problems where abstraction is *not* a useful technique, but here we investigate cases where it is.

```
WSet_N ⊎ Cex combinedAbstraction(Netlist N) {
    Trace  T(N);
    Wire_N  bad = ¬N.getProperty();
    WSet_F  bad_disj = ∅;

    for (int depth = 0;;) {
        if (⟨reached resource limit⟩)
            return T.abstr;

        bad_disj = bad_disj ∪ {T.insert(depth, bad)};

        if (T.solve(bad_disj)) {     – Found counter-example; refine abstraction:
            int n_flops = T.abstr.size();
            refineAbstraction(T, depth, bad);
            if (T.abstr.size() == n_flops)     – Abstraction stable ⇒ counter-example is valid:
                return T.getCex(depth);
        }else
            depth++;
    }
}

void refineAbstraction(Trace& T, int depth, Wire_N bad) {
    Cex                    cex = T.getCex(depth);
    Vec⟨WMap_N⟨lbool⟩⟩ sim = simulateCex(T.N, T.abstr, cex);       – 'sim[d][w]' = value if gate 'w' at frame 'd'

    WSet_N to_add;
    for all flops w not in T.abstr {
        for (int frame = 0; frame ≤ depth; frame++) {
            simPropagate(sim, T.abstr, frame, w, LBOOL_X);

            if (sim[depth][bad] == LBOOL_X) {
                – 'X' propagated all the way to the output; undo simulation and add flop to abstraction:
                for (; frame ≥ 0; frame−−)
                    simPropagate(sim, T.abstr, frame, w, cex.flops[frame][w]);
                to_add = to_add ∪ {w};
                break;
            }
        }
    }
    for w ∈ to_add
        T.extendAbs(w);
}

Vec⟨WMap_N⟨lbool⟩⟩ simulateCex(Netlist N, WSet_N abstr, Cex cex) {
    return ⟨ternary simulate counter-example 'cex' on 'N' under abstraction 'abstr'⟩
}

void simPropagate(Vec⟨WMap_N⟨lbool⟩⟩& sim, WSet_N abstr, int frame, Wire_N w, lbool value) {
    ⟨incrementally propagate effect of changing gate 'w' at time-frame 'frame' to 'value'⟩
}
```

Figure 2. *Main procedure.* Function ***combinedAbstraction***() takes a netlist and returns either (i) a counter-example (if the property fails) or (ii) the best abstraction produced at the point where resources were exhausted. We leave it unspecified what precise limits to use, but examples include a bound on the depth of the unrolling, the CPU time, or the number of propagations performed by the SAT solver. Function ***refineAbstraction***() will use the latest counterexample stored in T (by ***solve***(), if the last call was SAT) to grow the abstraction. Ternary (or X-valued) simulation is used to shrink the support of the counterexample. Abstract flops that could be removed from the support (i.e. putting in an X did not invalidate the counterexample) are kept abstract; all other flops are concretized. When simulating under an abstraction, abstract flops don't use the value of their input signal, but instead the value of the counterexample produced by the SAT solver (where the flop is a free variable).

Bench.	#Ands	#Flops	**Abstr. Size** (flops) New	New'	ABC	Hyb.	**Abstr. Time** (sec) New	New'	ABC	Hyb.	**Proof Time** (sec) New	New'	ABC	Hyb.	No Abs.
T0	57,560	1,549	2	4	2	6	0.1	0.1	0.3	0.5	0.1	0.1	0.1	0.2	0.4
T1	57,570	1,548	15	15	15	15	1.1	**0.7**	2.3	9.7	0.9	0.9	0.9	2.8	3.5
S0	2,351	1,376	**112**	157	174	–	**0.1**	0.3	2.0	–	8.7	129.7	**5.3**	–	21.2
S1	2,371	1,379	**136**	170	167	–	0.1	0.1	0.6	–	**57.8**	162.9	104.9	–	188.1
S2	3,740	1,526	**83**	123	113	187	0.3	**0.1**	0.6	26.0	**1.1**	37.1	11.8	106.7	4.3
D0	8,061	1,026	107	112	**106**	–	**3.0**	3.3	15.9	–	6.9	19.6	**4.9**	–	7.9
D1	7,262	1,020	139	139	139	139	1.2	1.2	4.4	**0.9**	0.3	0.3	0.3	2.9	0.6
M0	17,135	1,367	179	179	180	**178**	6.8	**6.3**	18.5	206.5	0.2	0.2	0.2	6.3	0.7
I0	1,241	1,104	59	57	**50**	–	0.5	**0.1**	0.6	–	2.0	1.9	**0.7**	–	5.8
I1	395,150	25,480	24	21	21	33	5.5	1.3	**1.1**	16.3	0.0	0.0	0.0	0.3	22.1
I2	5,589	1,259	45	**44**	51	–	1.5	**0.5**	1.5	–	6.2	**5.7**	6.8	–	18.0
I3	5,616	1,259	49	**47**	52	–	1.2	**0.4**	1.5	–	**5.9**	6.5	6.2	–	19.1
I4	394,907	25,451	79	**72**	100	–	64.3	**19.3**	30.9	–	**5.1**	15.0	17.9	–	–
I5	5,131	1,227	49	44	**38**	59	0.5	**0.1**	0.4	202.2	2.2	**0.2**	0.4	20.2	1.6
A0	35,248	2,704	**61**	68	95	81	1.8	**1.6**	6.3	6.9	18.9	**12.0**	35.7	18.3	43.2
A1	35,391	2,738	56	56	62	83	2.3	**1.7**	4.9	11.6	15.7	13.1	31.1	**6.9**	29.5
A2	35,261	2,707	8	8	18	24	0.1	0.1	0.2	0.8	0.0	0.0	0.0	0.2	0.6
A3	35,416	2,741	**59**	70	79	83	**2.2**	2.4	7.9	104.0	21.2	**11.5**	79.0	12.3	52.2
A4	35,400	2,741	**63**	65	67	101	2.5	**2.1**	4.4	34.4	**11.9**	20.0	36.2	12.1	34.6
F0	863,248	40,849	3	3	3	–	**1.0**	2.0	3.5	–	0.0	0.0	0.0	–	48.2
F1	863,251	40,850	4	8	4	–	**1.5**	4.7	7.0	–	0.0	2.2	0.0	–	100.6
F2	863,254	40,851	5	9	5	–	**3.9**	6.1	9.4	–	0.0	2.4	0.0	–	110.1

Table 1. *Evaluation of abstraction techniques.* Four implementations of hybrid counterexample- and proof-based abstraction were applied to 22 benchmarks of more than 1000 flops, all for which the property holds. In *New'*, PBA was only applied to the final iteration (to be closer to the ABC implementation). The first section of the table shows the size of the designs. The second section shows, for each implementation, the size of the smallest abstraction it produced that was good enough to prove the property. The third and fourth sections show the time to compute the abstraction, and the time to prove the property using interpolation based modelchecking, with the very last column showing interpolation on the original unabstracted design. Benchmarks with the same first letter denote different properties of the same design. Timeout was set to 500 seconds.

simulated in our new algorithm by delaying the PBA filtering until the final iteration (reported in column *New'*). This approach is often faster due to the fewer CBA refinement steps required, but there seems to be a quality/effort tradeoff between applying PBA at every step, or only once at the end. In particular for the *S* series, interleaved CBA/PBA resulted in significantly smaller abstractions. We have observed this behavior on other benchmarks as well.

The McMillan hybrid technique was improved by replacing BDDs with interpolation, which led to a significant and consistent speedup. However, our new method, and the similar techniques of ABC, still appear to be superior in terms of scalability. This is most likely explained by the expensive concretization phase of the older method, which requires the full design to be unrolled for the length of the counterexample.

The effect of an incremental implementation can be seen by comparing columns *New'* and *ABC*. We have observed that the speedup tends to be more significant for harder problems with higher timeouts.

The overall conclusion is that small abstractions help the proof-engine. However, there are cases where a tighter abstraction led to significantly longer runtimes than a looser one (although that effect did not manifested itself in this benchmark set). This can partly be explained by the underlying random nature of interpolant-based model checking, but it should also be recognized that replacing flops with PIs introduces more behaviors, which means the SAT-solver has to prove a more general theorem. Occasionally this can be detrimental, and offset the benefit of the reduced amount of logic that needs to be analyzed. Altogether, it emphasizes that abstraction should be used in good orchestration with other verification techniques.

7 Acknowledgments

This work was supported in part by SRC contracts 1875.001 and 2057.001, NSF contract CCF-0702668, and industrial sponsors: Actel, Altera, Calypto, IBM, Intel, Intrinsity, Magma, Mentor Graphics, Synopsys (Synplicity), Tabula, Verific, and Xilinx.

```
Trace::Trace(Netlist& N0) {
    N = N0;
    f2s[F.True] = S.True;
}

Wire_F Trace::insert(int frame, Wire_N w) {
    Wire_F ret = n2f[frame][w];
    if (ret == WIRE_UNDEF) {
        if      (w == N.True)      { ret = F.True; }
        else if (type(w) == PI)    { ret = F.add_PI(); }
        else if (type(w) == AND)   { ret = F.add_And(insert(frame, w_0), insert(frame, w_1)); }
        else if (type(w) == FLOP)  { ret = F.add_PI(); if (w ∈ abstr) insertFlop(frame, w, ret); }
        n2f[frame][w] = ret;
    }
    return ret ^ sign(w);                    – interpretation: (w ^ b) ≡ (b ? ¬w : w)
}

void Trace::insertFlop(int frame, Wire_N w_flop, Wire_F f) {
    Wire_F f_in = (frame == 0) ? ¬F.True : insert(frame−1, w_in);
    Lit p = clausify(f_in);
    Lit q = clausify(f);
    Lit a = act_lits[w_flop];
    if (a == LIT_UNDEF) {
        a = S.newLit();
        act_lits[w_flop] = a; }
    S.addClause({¬a, ¬p, q});
    S.addClause({¬a, p, ¬q});                – we've now added: a → (p ↔ q)
}

void Trace::extendAbs(Wire_N w_flop) {
    abstr = abstr ∪ {w_flop};
    for (int frame = 0; frame < n2f.size(); frame++) {
        Wire_F f = n2f[frame][w_flop];
        if (f != WIRE_UNDEF)                 – f is either undefined or a PI
            insertFlop(frame, w_flop, f);
    }
}
```

Figure 3. *Unrolling the netlist.* Method ***insert***() will recursively add the logic feeding w to netlist F. Flops that are concrete will be traversed across time-frames, but not abstract flops. Each flop that is introduced to F is given an *activation literal*. If this literal is set to TRUE, the flop will connect to its input; if it is set to FALSE, the flop acts as a PI. Activation literals are used to implement the proof-based abstraction, and to disable flops when the abstraction shrinks. At frame 0, flops are assumed to be initialized to zero. The purpose of ***extendAbs***() is to grow the abstraction by one flop, adding the missing logic for all time frames.

```
Lit Trace::clausify(Wire_F f) {
    Lit ret = f2s[f];            – map ignores the sign of 'f'
    if (ret == LIT_UNDEF) {
        if (type(f) == PI)
            ret = S.newLit();
        else if (type(f) == AND) {
            – Standard Tseitin clausification
            Lit x = clausify(f_0);
            Lit y = clausify(f_1);
            ret = S.newLit();
            S.addClause({x, ¬ret});
            S.addClause({y, ¬ret});
            S.addClause({¬x, ¬y, ret});
        }
        f2s[f] = ret;
    }
    return ret ^ sign(f);
}

bool Trace::solve(WSet_F f_disj) {
    Lit q = S.newLit();
    S.addClause({¬q} ∪ {clausify(f) | f ∈ f_disj});

    assumps = {q} ∪ {act_lits[w] | act_lits[w] != LIT_UNDEF && w ∈ abstr};

    bool result = S.solve(assumps);
    if (result) ⟨store SAT model⟩
    else        abstr = abstr \ {w | type(w) == FLOP && w ∉ S.conflict};      – this line does PBA

    S.addClause({¬q});     – forever disable temporary clause
    return result;
}

Cex Trace::getCex(int depth) {
    return ⟨use maps 'n2f' and 'f2s' to translate the last SAT model
            into 0/1/X values for the PIs and Flops of frames 0..depth⟩
}
```

Figure 4. *SAT-Solving.* Method ***clausify***() translates the logic of F into CNF for the SAT-solver using the Tseitin transformation. The above procedure can be improved, e.g., by the techniques of [4, 12]. Method ***solve***() takes a disjunction of wires in F and searches for a satisfying assignment to that disjunction. Because only unit assumptions can be passed to ***solveSat***(), a literal q is introduced to represent the disjunction, and a temporary clause is added. Disabling the clause afterwards will in effect remove it. The activation literals of the current abstraction are passed together with q as assumptions to ***solveSat***(). The SAT-solver will give back either a satisfying assignment (stored for later use by ***getCex***()), or a conflict clause expressing which of the assumptions were used for proving UNSAT. This set is used to perform PBA. In computing *assumps*, we note that "&& $w \in$ *abstr*" is necessary if PBA has shrunken the abstraction. In the experimental section, a variant (column *New'* in Table 1) is evaluated where PBA is not applied inside ***solve***(). The set of redundant flops is still computed as above, and remembered. When the resource limit is reached, those flops that were redundant in the final UNSAT call are removed. In essence, the variant corresponds to an incremental CBA implementation with a final trimming of the absraction by PBA.

References

[1] N. Amla and K. McMillan. **A Hybrid of Counterexample-based and Proof-based Abstraction**. In *FMCAD*, 2004.

[2] N. Amla and K. McMillan. **Combining Abstraction Refinement and SAT-based Model Checking**. In *TACAS*, 2007.

[3] P. Chauhan, E. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang. **Automated Abstraction Refinement for Model Checking Large State Spaces Using SAT-based Conflict Analysis**. In *FMCAD*, 2002.

[4] N. Een, A. Mishchenko, and N. Sorensson. **Applying Logic Synthesis for Speeding Up SAT**. In *SAT07*, volume 4501 of *LNCS*, 2007.

[5] Niklas Een and Niklas Sörensson. **The MiniSat Page**. *http://minisat.se*.

[6] Berkeley Logic Synthesis Group. **ABC: A System for Sequential Synthesis and Verification**. *http://www.eecs.berkeley.edu/~alanmi/abc/*, v00127p.

[7] A. Gupta, M. Ganai, Z. Yang, and P. Ashar. **Iterative Abstraction Using SAT-based BMC with Proof Analysis**. In *ICCAD*, 2003.

[8] R. P. Kurshan. **Computer-Aided-Verification of Coordinating Processes**. In *Princeton Univ. Press*, 1994.

[9] B. Li and F. Somenzi. **Efficient Abstraction Refinement in Interpolation-Based Unbounded Model Checking**. In *TACAS*, 2006.

[10] K. McMillan. **Interpolation and SAT-based Model Checking**. In *CAV*, 2003.

[11] K. McMillan and N. Amla. **Automatic Abstraction without Counterexamples**. In *TACAS*, 2003.

[12] D. Vroon P. Manolios. **Efficient Circuit to CNF Conversion**. In *SAT*, 2007.

[13] D. Wang, P. Jiang, J. Kukula, Y. Zhu, T. Ma, and R. Damiano. **Formal property verification by abstraction refinement with formal, simulation and hybrid engines**. In *DAC*, 2004.