Using Speculation for Sequential Equivalence Checking

Robert Brayton **Niklas Een** **Alan Mishchenko**

Department of EECS, University of California, Berkeley

brayton@eecs.berkeley.edu niklas@een.se alanmi@eecs.berkeley.edu

Abstract

An improved method for speculative reduction is proposed and applied to (suspected) hard verification problems. Several variations of the algorithm were tested: (a) applying speculation initially to the original problem; (b) applying speculation after simplification, before our regular model checker, super_prove *is applied, as well as (c) using different filters to reduce the number of speculated equivalences tried. On the benchmarks coming from sequential equivalence checking, the speculation-first strategy with filtering proved to be faster than* super_prove. *On other benchmarks that may have come from property checking,* super_prove *is found superior.*

1 Introduction

The application of formal verification in industry has increased substantially in the last decade. Formal verification can be classified as a) property checking or b) sequential equivalence checking (SEC).

Property checking has been more of a research focus and its use has increased significantly. However, SEC is probably still the main challenge for applying formal verification to sequential circuits.

In the last decade, state-of-the-art property and equivalence checking methods have become increasingly similar, with verification engines such as synthesis, induction, interpolation, BDD and property-directed reachability applied to both classes of problems. In most of the publicly available verification benchmarks, it is usually not known whether the source is property checking or SEC. Thus the advanced present-day software uses a common approach for both types of problem.

This paper focuses on SEC to try to take advantage of the special characteristics of such problems.

In sequential equivalence checking (SEC), two sequential circuits are compared. Typically, one is derived from the other by logic synthesis or manual intervention. SEC problems can be specified in two ways:

- A single circuit is given wherein the two original circuits have been mitered together by forming XORs of pairs of related POs.
- Two circuits are given separately, or in the form of a dual-output miter, which lists pairs of related outputs in the same file.

In general, the second representation is preferred because the clean separation between the two circuits that are being checked for equivalence can be exploited by a SEC engine. Moreover, given the second representation, it is trivial to derive the first, but not vice versa.

In this paper, however, we focus on the first representation since most of the available benchmarks have already been mitered.

The SEC problem has an advantage over property checking because it is expected that there are many pairs of equivalent signals, where one signal of the pair is in the first copy and another is in the second copy. Although, there may be equivalent pairs where both signals are in the same copy, these are typically not useful in proving SEC.

Our main approach to SEC is through the use of speculative reduction [2][3]. In this, the circuit is simulated on the reachable state set until the equivalence classes of signals generated are no longer being refined. For this simulation, the notion of *rarity simulation* is used, which is explained in Section 3.

After simulation, the remaining equivalence classes are used to form a speculatively reduced model (SRM), in which all fanouts of signals in an equivalence class are transferred to a representative signal of that class. Then XORed pairs of signals in the class are formed in a round-robin fashion. These are made (pseudo) POs of the SRM, and serve as additional proof obligations.

If, in subsequent solving of the SRM, any the pseudo-POs is disproved, the SRM is invalid, and the cex found is used to further refine the equivalence classes, leading to a new SRM. If a counterexample (cex) has been found to an original PO, the equivalence is disproved and SEC terminates. However, if all POs are proved, then the original SEC problem is proved UNSAT, because the original POs are proved to be part of the class of signals equivalent to the constant 0.

The SRM circuit produced as shown above has several characteristics. Typically, it has many outputs, and it is expected that all are UNSAT, so finding a cex is given higher priority in the ensuing algorithms. Second, the SEC problem is not proved UNSAT unless all speculated equivalences are valid. This means that even if there exist only one invalid equivalence, the SEC problem is not solved. Therefore, we should try to speculate only on those equivalences which are ‘relevant’ to proving SEC.

These characteristics dictate a different orchestration that can be done to speed up solving SEC.

The contributions of this paper are as follows.

1. Our implementation of speculative refinement is described.
2. Rarity simulation used to quickly refine equivalence classes is presented.
3. Methods to reduce the number of proof obligations are discussed:
   - isomorphic ones are detected and removed;
   - sequentially constrained limited resource induction is used to remove others;
   - an optional filter is used to limit equivalences to those containing a FF.
4. A two-phase method is used to prove or disprove the remaining equivalence classes.

This paper is organized as follows. Section 2 reviews *speculative reduction*. Section 3 describes *rarity simulation*. Section 4 gives an overview of a general proof engine, *super_prove*. Section 5 outlines our verification flow for SEC, and Section 6 provides some experimental results. Section 7 concludes with future work that may help further improve the SEC engine.

2 Improved Speculative Reduction

Our method of speculative reduction proceeds in the following sequence. It has several enhancements, which we comment on later.

1. Rarity simulation (discussed in Section 3) is done until satuation, to create a set of candidate equivalence classes, i.e classes of signals speculated to be equivalent on the set of reachable states.
2. These are filtered optionally according to several criteria, *none*, *f, g,* and *ab*.
3. The speculated miter (SRM) AIG is created. This has multiple POs which are proof obligations stating that the speculated equivalences are valid. The main body of the sequential AIG is reduced by assuming that the equivalences are valid.
4. Proof obligations are reduced by fast induction and isomorpism detection.
5. Proof of the resulting multi-output SRM is attempted.
   - If a counter-example (cex) is found to any of the speculated outputs, it is used to refine the equivalence classes and a new SRM is created.
   - If a cex is found to an original output, the SEC problem is declared SAT.
6. The remaining outputs are lumped together and attempted all at once using *super-prove*.
7. If this times out, then the outputs are attempted individually in two stages using s*uper_prove.*

Below we comment on the steps that differ from the standard implementation of speculation.

Filters. There are three types of filters defined and used optionally to reduce the number of equivalence classes: *none, g, f, ab*. Filter *none* refers to not eliminating any classes. Filter *g* refers to eliminating any class that does not have at least one FF. Filter *f* refers to eliminating any class that does not have at least two FFs, and filter *ab* is used when the 'A' side and the 'B' side of the SEC problem are known. Then any class is eliminated which has signals in one side only. The idea of using filters on the equivalence classes generated is to cut on the number of proof obligations because the SRM cannot be proved valid until all speculated equivalences are proved. The filters are used to focus on 'relevant' equivalences.

Fast induction. This is motivated by the characteristic of a SEC problem: it is expected that each PO is UNSAT. Thus it is reasonable to try to prove one PO, assuming that all the rest are UNSAT. This sets up a constrained verification problem, where we try to prove the designated PO using induction while using the other POs as constraints. If the induction proof succeeds in proving the PO UNSAT, then we *eliminate* the designated PO and proceed to the next PO. If the proof leads to SAT (which is very rare), then we use the generated cex to refine the equivalences and derive a new SRM. If the proof times out, then the designated PO is marked, but left on the list, and the next PO is attempted. By eliminating proved POs, we do not get into a cyclical argument where we assume one thing to prove another and then assume the other to prove the first. Experiments show that this fast induction can quickly eliminate a substantial percentage of Pos created during speculation.

Using isomorphism. If a group of outputs are detected to be structurally isomorphic, then it is enough to prove one of them [23].

Proving all outputs at once versus attempting each output separately. Several problems can be proved quite easily once the speculated outputs are valid. In trying to prove all outputs at once, we may be able to eliminate bad outputs, by looking for a cex on any of the outputs. On the other hand, having to deal with all outputs at once may obscure a cex for a particular output, which can be disproved easily if the logic for just that output is kept and the other logic is eliminated. For example, if one output has a hard cex at cycle 50 and another one has an easy cex at cycle 75, then BMC would timeout at 50 and it would not be known that the SRM is invalid. On the other hand, dealing with all outputs at once allows us to find an easy cex more quickly.

Two-stage proof step. The idea of proving the set of remaining outputs individually in two stages is to look for a cex first. If one is found, then time is not wasted in proving other outputs. We tried using different time-outs for this but eventually settled on a 10-second and 100-second two-phase approach as the most practical. One strategy that works well is, if a cex is found when trying to prove Output *k*, then after a new SRM is produced, the next proof attempt starts at Output *k* (even though the number of outputs might have changed).

3 Rarity Simulation

It has been found to be extremely important to have a simulation method that is fast but capable of creating a set of equivalences that is closer to a valid set. Experience with random simulation and constrained simulation was that the first saturated too fast leaving many invalid speculations, and the second was too slow. This led us to believe that speculative reduction was not applicable to large industrial problems.

However, it was suggested by the *SixthSense* team at IBM that we should do something like the *rarity simulation,* which we describe below. This made speculative reduction a significant contributor to our formal verification approach and probably was the most significant factor in ABC's *super_prove* [14][8] winning the latest hardware model checking competition [10].

Sequential simulation begins in the initial state and proceeds computing states reachable from the initial one, by applying sequences of primary input values in the subsequent time steps. Random simulation generates random values at the primary inputs and applies them for a varying number of cycles. It is repeated for a fixed number of rounds, or until some other criterion is reached (e.g., there is no more refinement of equivalence classes). Random simulation is simple and easy to implement, but saturates quickly because the random stimulus does not take into account properties of the sequential circuit.

Constrained random simulation is used to augment random simulation when it saturates too quickly or when simulation under user-specified constraints is required. Constrained random simulation can find input sequences that visit some special states (states where constraints hold or states where interesting events happen, for example, states where hard-to-refine equivalence classes are refined), but it mandates the use of a SAT solver or an ATPG engine, and therefore is more elaborate to implement and slower than random simulation.

We propose a variation of guided random simulation, which uses heuristics to guide selection of states, from which simulation is allowed to continue. The primary input values are still selected randomly, as in the case of regular random simulation, but the new current states are selected among the next states using a criterion called *rarity*. Rarity of a state is a measure showing how often this state, or flop values appearing in this state, appears during random simulation. To facilitate collecting the required rarity information, we split the state vector into groups of flops of a fixed length, in the natural order of the flops' appearance in the design. The default parameter used in our implementation is 8 flops per group, beginning from the first flop till the last. If there are any left-over flops not included in the last group, they are ignored.

For each group of flops, the rarity-based simulator collects statistics showing how many times a specific value of these flops has appeared in the next states reached by simulation so far. The statistics are represented as integer counters, one per each value of each group. For example, if we have N flops and use K-flop groups, we need $2^K N / K$ counters. Now each next state reached can be characterized by a weight equal to the sum of inverse values of the counters corresponding to specific flop group values appearing in this state. The weight is greater for those reached states whose flop group values appear less frequently.

Now the reached states are sorted by their weight, and only a fixed number of states with the highest weight values are used for simulation in the next iteration. The default parameters used in our implementation are: the simulation begins from the initial state and proceeds with 50 64-bit machine words of random primary input data (3200 patterns are simulated in bit-parallel fashion); the weights are recomputed after 20 timeframes. Then 50 states with the highest weight are selected among 3200 resulting patterns, and simulation is repeated from these states for another 20 timeframes.

Experimental results have shown that this works well for refining candidate equivalence classes of sequentially equivalent nodes. We speculate that this is because the rarity-based simulation navigates nicely through the complex state spaces and selects rare states to be used as simulation seeds. This allows for interesting sequential behavior to manifest. Moreover, this rarity-based simulation may natively handle the reset phenomenon which plagues regular random simulation. Indeed, if a reset signal is part of the design, random simulation will force half of the states seen in the next cycle to differ from the initial state, so that after 50 cycles we will have only seen 25 non-initial states, and among those most will be states that can be reached in depth 1 from the initial state. The rarity-directed heuristic will pick deeper states, as the new initial states for future simulation rounds.

4 Overview of *super_prove*

The name *super_prove* is given to our model checker, which entered HWMCC'11 and won the first place in the *combined* and *UNSAT* categories.

The algorithm implemented in *super_prove* uses a hybrid concurrent approach where several model checking (MC) engines are run concurrently.

In ABC, the following MC engines are available:

1. Random or rarity simulation
2. Semi-formal simulation
3. Bounded model checking (BMC) [13]
4. BDD-based reachability [6][19]
5. Property directed reachability (PDR) [4]
6. Interpolation [12][9]
7. Synthesis:
    a. rewriting [9]
    b. retiming [11]
    c. sequential signal correspondence [20]
    d. phase abstraction [21]
    e. temporal decomposition [18]

8. Abstraction: [7]
    a. counterexample-based (CB) [15]
    b. proof-based (PB) [16] [17]
9. Speculation [2][3]

An engine can be classified as a (i) *verification engine*, that either finds a bug-trace or proves the property (engines 1-6), or a (ii) *transformation engine*, which attempts to reshape or decompose the problem into one or more simpler problems (engines 7-9).

Verification engines can be classified further into *complete* ("proof-producing/bug-finding", Engines 4-6) and *incomplete* ("bug-hunting only", Engines 1-3). Transformation engines can be either *equivalence preserving* (engine 7) or *abstracting* (8 and 9).

Once abstraction has been applied, bug-traces may be spurious and only proofs of unsatisfiability are conclusive. However, spurious traces can be used to refine the current abstraction until the property, if true, can be proved.

In *super_prove,* shown in Figure 1, a number of MC engines are used concurrently to prove of disprove a miter, which is denoted by the term *c-verify*. Methods separated by || in boxes are run concurrently; a solid arrow means the result is passed on from a terminating engine; a dotted arrow means that *c_verify* from the upper level continues in parallel with other engines that are started later. Terms *c_abstract,* and *c_speculate* are labeled with '*c*' because the refinements in them are done concurrently. In practice, the number of cores is limited, so as soon as a new box starts, the previous computation is terminated.

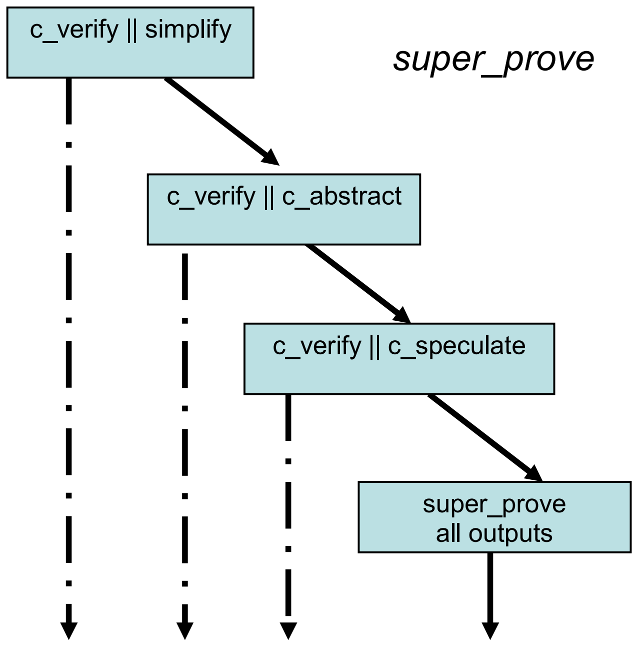

Figure 1. An outline of the hybrid concurrent MC algorithm, *super_prove*.

If at any time an engine produces a definitive result, all processes are terminated and the proof is complete.

The idea of doing speculation first on the original problem is motivated by wanting to find as many useful equivalences as possible, because any synthesis destroys equivalences that may be useful.

5 Other Methods

The method *ss* differs from *super_prove* only in that speculation is applied initially to the original AIG followed by *super_prove*. Method *ssm* differs from *ss* in that simplification is done first on the file followed by speculation and then *super_prove*. Variations using filtering on *ss* and *ssm* allow filtering using the '', 'g' or 'f' options. These are referred to in Section 5 as *ss*(''), *ss*('g') *ss*('f'), *ssm*(''), *ssm*('g'), and *ssm*('f')

6 Experimental Results

We compared the new verification flow *ss* described above with *super_prove* on hard MC benchmarks.

The first part of the table contains selected IBM benchmarks. The last part contains benchmarks from the model checking competition HWMCC'11 [10], which were either not solved or uniquely solved by *super_prove* in the 900 seconds allocated for each problem (i.e. no other entrant solved the problem).

The results are shown in Table 1. In this experiment, we are only run examples where we suspect that the problem is a SEC problem, since otherwise we do not expect that trying speculation first to be superior to *super_prove,* which tries abstraction first after simplification. The idea is that for a user, who might know what kind of problem is being solved, the *speculation-first* methods can be chosen for SEC problems and the *super_prove* method can be chosen otherwise. The following speculation-first methods were tried, *ss*, *ssm*(''), *ssm*('g') and *ssm*('f')

The *ss* method was run mainly with the null filter option ('') being given, but the 'g' option *ss*('g') was compared with this on 4 examples. Since the 'g' option proved to be marginal at best, we did not experiment further with *ss*('g') as well as *ss*('f'). Also, since we did not have access to reasonably hard SEC problems where the two circuits are given separately, the 'ab' option has not been tested yet.

Other methods were based *on* ss(''), where simplification was done first before *ss* was called. These were *ssm*(''), *ssm*('g') and *ssm*('f'). where various filter options were given.

Table 1 shows the results on 23 benchmarks selected from the HWMCC11 benchmark set. These were chosen to be reasonably hard where there was a possibility that they might be SEC problems. We know that the problems beginning with 'bob' are indeed SEC problems. On those we expected the SEC oriented approach would bear fruit. The first 3 of these were solved already by *simplify*, so only *ss*('') suffered and the others got the same run-times. On the remaining 'bob' examples, the method *ssm*('g') seems to be a good option and demonstrated that doing speculation before abstraction was a good strategy. That 'g' seemed better that '' might be explained by the fact that there were many speculated equivalences and filtering out some was effective. However, filtering out too many like 'f' does can be detrimental.

On the next 3 examples only *ss*('') and *sp* were compared. Since *ss*('') did not provide any advantage, it was not run on the remaining examples, because we concluded that initial simplification was the method of choice. On the remaining 11 examples, it might be that none of them were SEC problems, and this might be showing up in the fact that *sp* was basically the better method although on a few examples, doing speculation first reduced the runtime.

7 Conclusions and Future Research

A key ingredient of model checking is the use of speculation. All the methods tried in this paper used the same implementation of speculation based on the advanced features such as rarity simulation and improved in other ways as described in Section 2.

We postulated that doing speculation first on SEC problems might be a good strategy. We have described a limited set of experiments comparing variations of this idea against *super_prove* which uses the strategy of first simplifying and then doing abstraction followed by speculation. Various filters were tried to trim down the redundant speculated equivalences and working only on the "relevant" ones .

We described an improved version of speculation which was used in the model checker *super_prove*.

The experimental results indicate the following:

1. Even if a problem is known to be SEC problem, it is still a good idea to simplify the problem first before trying either abstraction or speculation.
2. The filter option, '*g*', is indicated to be a good strategy on SEC problems while 'f' seems to filter out too many useful equivalences.

These are only impressions after a limited number of experiments, and a more definitive set of experiments should be done when we assemble a larger suite of hard SEC problems. Also the 'ab' strategy described in this paper remains to be experimented with.

An interesting strategy for the future might be to initially estimate or even compute how many initial equivalence classes are found by rarity simulation. This can be reasonably fast because rarity simulation converges quickly. For examples with many classes, one could try a speculation-first strategy, like *ssm*('g'). On the other hand, we could just try *sp* and *ssm* in parallel if there are many processors available.

Acknowledgements

This work was partly supported initially by SRC contract 1875.001, NSA grant "Enhanced equivalence checking in crypto-analytic applications". We thank industrial sponsors of BVSRC: Altera, Atrenta, Cadence, Calypto, IBM, Intel, Jasper, Microsemi, Real Intent, Synopsys, Tabula, and Verific, for their support. The authors acknowledge stimulating discussions with IBM's SixthSense team, which led to developing rarity simulation and other methods presented in this paper.

References

[1] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G. Janssen, "Scalable sequential equivalence checking across arbitrary design transformations", *Proc. ICCD'07*, pp. 259-266.

[2] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman. "Exploiting suspected redundancy without proving it". *Proc. DAC'05*, pp. 463-466.

[3] H. Mony, J. Baumgartner, A. Mishchenko, and R. Brayton, "Speculative reduction-based scalable redundancy identification", *Proc. DATE'09*, pp. 1674-1679.

[4] A. R. Bradley, "k-step relative inductive generalization", http://arxiv.org/abs/1003.3649

[5] A. R. Bradley and Z. Manna, "Checking safety by inductive generalization of counterexamples to induction", *Proc. FMCAD'07*.

[6] F. Somenzi, *BDD package CUDD*. http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html

[7] N. Een, A. Mishchenko, and N. Amla, "A single-instance incremental SAT formulation of proof- and counterexample-based abstraction". *Proc. FMCAD'10*.

[8] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool", *Proc. CAV'10*, Springer, LNCS 6174, pp. 24-40.

[9] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis", *Proc. DAC '06*, pp. 532-536.

[10] http://fmv.jku.at/hwmcc11/

[11] A. P. Hurst, A. Mishchenko, and R. K. Brayton, "Fast minimum-register retiming via binary maximum-flow", *Proc. FMCAD '07*.

[12] K. L. McMillan, "Interpolation and SAT-based model checking". *Proc. CAV'03*, pp. 1-13.

[13] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. "Bounded model checking using satisfiability solving", *Proc. Formal Methods in System Design (FMSD)*, vol. 19(1), Kluwer 2001

[14] Berkeley Verification and Synthesis Research Center. *ABC: A System for Sequential Synthesis and Verification*. http://www.eecs.berkeley.edu/~alanmi/abc/

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

[16] K. McMillan and N. Amla, "Automatic abstraction without counterexamples". *Proc. TACAS'03*.

[17] A. Gupta, M. Ganai, Z. Yang, and P. Ashar. "Iterative abstraction using SAT-based BMC with proof analysis". *Proc. ICCAD'03*.

[18] M. L. Case, H. Mony, J. Baumgartner, R. Kanzelman. "Enhanced verification by temporal decomposition". *Proc. FMCAD'09*, pp.17~24

[19] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan and D. L. Dill, "Symbolic model checking for sequential circuit verification", *IEEE TCAD*, vol. 13(4), 1994, pp. 401-424.

[20] A. Mishchenko, M. L. Case, R. K. Brayton, and S. Jang, "Scalable and scalably-verifiable sequential synthesis", *Proc. ICCAD'08*, pp. 234-241.

[21] P. Bjesse and J. Kukula, "Automatic generalized phase abstraction for formal verification", *Proc. ICCAD '05*.

[22] P.-H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, J. Long, "Smart Simulation Using Collaborative Formal and Simulation Engines," *Proc. ICCAD'00*.

[23] A. Mishchenko, N. Een, R. Brayton, M. Case, P. Chauhan, and N. Sharma, "A semi-canonical form for sequential AIGs", *Submitted to IWLS'12*.

Table 1. Comparing solve times for methods with different options.

NAME	ss('')	sp	ssm('')	ssm('g')	ssm('f')
*bobsm38584	timeout	10.04	9.92	-	-
*bobsmfpu	1225.14	27.37	27.59	-	-
*bobsmmips	timeout	168.82	169.35	172.78	172.03
bobsmhdlc	336.26	**	227.41	**46.63**	117.77
bobsmhdlc1	35.40	362.77	**30.35**	39.70	56.38
bobsmhdlc2	117.23	248.92	93.74	**35.13**	71.65
bobsmhdlc3	**45.45**	*	97.44	63.96	1103.94
bobsmminiuart	timeout	timeout	timeout	-	-
bobsmoci	36.12	60.68	**13.36**	25.22	52.00
pdt_qis10x6p1	151.00	**28.00**	-	-	-
pdt_qis8x8p1	66.00	**12.00**	-	-	-
pdt_qis8x8p1	582.00	**568.00**	-	-	-
6s0	-	timeout	**895.98**	timeout	timeout
6s21	-	**405.66**	timeout	timeout	timeout
6s51	-	**135.07**	138.80	166.81	137.99
6s9	-	**143.78**	timeout	timeout	264.24
*bjrb07amba9andenv	-	72.19	**67.51**	71.84	70.36
pdtfifo1to0	-	917.85	881.22	**429.29**	timeout
pdtswvsam4x8p4	-	41.20	**29.89**	209.37	93.72
pdtswvsam6x8p4	-	timeout	timeout	timeout	timeout
pdtswvtma6x6p2	-	**172.39**	244.46	267.90	268.94
pj2017	-	**116.98**	318.81	229.91	234.46
tp_pib_w_0	timeout	**39.79**	99.55	56.00	312.17

All times are reported in seconds. **Bold** shows the method with the least time among those tried.
Dash (-) means that it was not tried.
* sm38584, smfpu, and smmips all solved by initial simplify.
** smhdlc2 while executing 'sp', an error occurred. It is being investigated.
timeout indicates that a time-out of 1000 sec occurred.