# Variable Time-Frame Abstraction

Alan Mishchenko Niklas Een Robert Brayton
Department of EECS, University of California, Berkeley
{alanmi, brayton}@eecs.berkeley.edu niklas@een.se

Jason Baumgartner Hari Mony Pradeep Nalla
IBM Systems and Technology Group
{baumgarj, harimony}@us.ibm.com pranalla@in.ibm.com

## Abstract

Verification benefits from removing logic that is not relevant for a proof. Techniques for doing this are generally known as *localization abstraction*. Abstraction is often performed by selecting a subset of gates to be included in the abstracted model; the signals feeding into this subset become unconstrained cutpoints. In this paper, a novel approach to gate-level abstraction is presented, which allows for including gates in some but not all time-frames of the abstracted model. The "variable time-frame" approach is motivated by the hope of building a more scalable abstraction, because the complete logic of the abstracted model in not used in each time-frame of the unrolling. Furthermore, a similar method could make BMC and interpolation-based model checking more scalable, by considering only a subset of gates belonging to the cone-of-influence of the property. Preliminary results suggest that it may be a promising approach.

## 1. Introduction

Localization abstraction plays an important role in reducing the complexity of formal verification. The known abstraction methods can be classified as follows:

- Automatic vs. manual
- SAT-based vs. BDD-based vs. other
- Proof-based vs. counter-example-based vs. hybrid
- Flop-level vs. gate-level
- Fixed time-frame vs. variable time-frame

The first criterion asks whether abstraction is performed automatically by a tool or manually by a verification engineer. Next, the chosen abstraction can be based on different computation methods; SAT-based techniques have largely replaced BDD-based or other methods. For SAT-based methods, abstraction refinement can rely on computing a sequence of UNSAT cores [15] or counter-examples [16] or by applying a hybrid approach, which combines proofs with counter-examples [1][12].

In terms of the granularity of the abstraction algorithm, abstraction can be flop-level [16][12] or gate-level [3], depending on whether it is constructed using entire next-state logic cones or individual gates as primitives. The final characteristic has to do with how an abstraction is checked: a gate may be included in all time-frames of the unrolled abstracted model (fixed time-frame) or in some time-frames and not in others (variable time-frame).

Using the above taxonomy of abstraction techniques, the method presented in this paper can be classified as follows:

- Automatic
- SAT-based
- Hybrid
- Gate-level
- Variable time-frame

To our knowledge, the proposed approach is the first work on variable time-frame abstraction (VTA).

The use of VTA reduces the size of the bounded unrolling of the abstracted model and facilitates deeper BMC [6] to test validity of the abstraction. As a result, refinement is more scalable and derived abstractions are more precise.

Experiments show that, on average, SAT instances can be reduced about 50% by not including each gate used in the abstraction into each time-frame of the unrolling.

The rest of the paper is organized as follows. Section 2 describes the background. Section 3 describes the algorithm. Section 4 reports experimental results. Section 5 concludes the paper and outlines future work.

## 2. Background

The standard definitions related to Boolean algebra, logic networks, And-Inverter Graphs (AIGs), SAT solving, UNSAT cores, etc. are assumed to be known to the reader.

Only definitions specific to this paper are listed below.

A safety property is represented by a sequential AIG. We call this sequential AIG a miter, even if the problem domain is property checking, not equivalence checking.

An *AIG gate* refers to any object that found in the AIG: the constant node, a primary input, a flop, an internal AND node, and a primary output.

An *abstraction* of depth K is a subset of gates of the sequential miter, such that if only this subset is included in the bounded unrolling of the miter and the rest are treated as unconstrained primary inputs, the resulting bounded property is "true" in the first K time-frames of the unrolling. For scalability of subsequent verification, this subset of gates should be as small as possible – though given the heuristic nature of the abstraction-refinement process, generally the resulting abstraction may include some gates which are unnecessary.

The notions, "an abstraction of the miter" and "an abstracted model", are used interchangeably.

An *included (excluded) gate* is a gate used (not used) in the abstracted model. In the present work, the constant node and the property output are always included. Included also can be some primary inputs, flops, and internal nodes.

An *abstraction pair* (AP) is an ordered pair of integer numbers (an AIG gate ID and a time-frame ID), which represents a given AIG gate in a given time-frame.

Given a bounded unrolling of a sequential miter of an abstracted model, an *UNSAT core* $C_i$ of frame K is a set of APs such that the SAT instance containing the following constraints is unsatisfiable:

- all the gates listed in the APs of the UNSAT core;
- constraints representing the initial state of the flops in the starting frame, if they are part of the UNSAT core;
- statement that the property output fails in frame K.

In this paper, *VTA* stands for the proposed method called "variable time-frame abstraction". The same acronym is used for an abstraction produced by this method. In this sense, a VTA of depth K is a set of APs, which should be included in the unrolling of the abstracted model to guarantee that the property output does not fail in any of the time-frames from 0 to K, inclusive.

A VTA can be given in two forms:

- As a set of UNSAT cores, one for each time-frame.
- As a union of APs included in the UNSAT cores.

# 3. Algorithm

## 3.1 The main idea

The main idea of the proposed method is to learn what gates should be included in the abstraction by analyzing UNSAT cores of the bounded unrolling of the original sequential miter and of its abstraction under construction.

Given UNSAT cores for the first few frames, the analysis results in a guess what AIG gates should be included in the UNSAT core of the next frame. If the guess is correct, the next frame's SAT instance is indeed UNSAT and its refined core is extracted and added to the abstraction under construction. If a guess is wrong, a refinement is performed, resulting in a better guess.

The abstraction engine terminates after exploring a fixed number of frames, when a resource limit is reached, for example, when there was no refinement to the abstraction in several recent frames. An abstracted model is returned.

If a true counter-example is discovered at any time, computation terminates, and the counter-example is returned. If the SAT solver runs out of resources, the computation is stopped and "undecided" is returned.

## 3.2 Abstraction framework

In this section, the method is described in detail. The pseudo-code is listed in Figure 3.2.

Computation of the UNSAT cores begins by working on the original miter. A bounded unrolling is performed and a BMC instance is constructed, containing the first S frames (S is a user-controllable parameter set to 5 by default).

If the problem is UNSAT in the first S frames, UNSAT cores of these frames are collected and saved.

For each frame after the first S, in their natural order, a SAT-based exploration is performed to determine what new gates should be included into the unrolling, beginning in the first frame and ending in the last frame, for the resulting problem to be UNSAT in the last frame.

```
aig deriveAbstraction (
   aig A,          // A is a sequential miter
   params P )   // P is a set of parameters
{
    initialize the set of UNSAT cores N to empty;
    // create bounded unrolling for S frames
    for each time-frame F from 0 until S-1 {
        if ( property output is SAT in frame F )
            return the counter-example;
        if ( property output is UNSAT in frame F )
            add the UNSAT core to N;
    }
    // derive next UNSAT cores by guessing and refinement
    for each time-frame f from S until Fmax {
        guess the UNSAT core Cf of frame f
            using UNSAT cores of previous P frames;
        while ( the resulting SAT instance is satisfiable ) {
            if ( it is a true counter-example )
                return the counter-example;
            perform refinement by adding more logic;
        }
        if ( the resulting SAT instance is UNSAT )
            add the UNSAT core to N;
    }
    // convert the set of UNSAT cores into the abstraction
    return abstraction derived by including all gates
        appearing in the APs of the UNSAT cores of N;
}
```

**Figure 3.2. Abstraction framework.**

In each new time-frame (S, S+1, etc), this exploration is performed in two steps:

- Initially, the UNSAT cores of P previous time-frames are considered (P is a user-controllable parameter set to 4 by default). Every abstraction pair (AP) included in each of these UNSAT cores is lifted by a fixed number of frames to create an approximation of the UNSAT core of the current frame. For example, while at frame 10, it may be observed that the UNSAT core of frame 9 contains APs (244, 9) and (205, 7). These are lifted by one frame to become APs (244, 10) and (205, 8), which are now included in the approximation of the UNSAT core of frame 10. If a lifted AP is already present in the approximation, it is ignored.
- Next, the approximation of the UNSAT core is loaded into the SAT solver and an incremental SAT run is made attempting to prove that the property output never fails in the current frame. If it returns UNSAT, which occurs often in practice, the approximation indeed contained an UNSAT core, which is extracted and saved. If the result is SAT, the counter-example is used to compute a refinement of the abstraction, as shown in Section 3.6 below. (Note that a counter-example may exist because the UNSAT core is not complete, even if abstraction is already precise.) The SAT runs and the refinement runs are iterated until the result is UNSAT.

If a true counter-example is not encountered after completing a fixed number of time-frames (Fmax), or if a resource limit is reached (such as runtime, conflict limit, etc), the resulting set of UNSAT cores is transformed into an abstraction of the sequential miter. This transformation finds a union of all gates appearing in any of the APs of the UNSAT cores. These gates become included gates of the abstraction. A new AIG representing the abstracted model is derived and returned to the caller.

### 3.3 Priority-based abstraction refinement

This subsection describes the counter-example-based abstraction refinement. This method is largely orthogonal to VTA and compatible with other abstraction algorithms. The pseudo-code is listed in Figure 3.3.

```
logic gates performAbstractionRefinement (
   aig A,          // A is a sequential miter
   aig C,          // combinational unrolling of an abstraction of A
   params P )  // P is a set of parameters
{
    divide the set of PIs of C into RPIs and PPIs;
    perform traversal to compute distances of the PPIs to the PO;
    assign priority to PIs using priority assignment rules;
    perform traversal to compute priorities of all gates of C;
    perform traversal to select a subset of PPIs;
    return logic gates driving the PPIs of the subset in the miter;
}
```

**Figure 3.3. Priority-based abstraction refinement.**

Consider a multi-level combinational circuit with a single property output (PO) and two types of primary inputs (PIs), called *real PIs* (RPIs) and *pseudo PIs* (PPI). Consider a complete assignment of the PIs, which forces PO to "fail".

A refinement algorithm finds a minimal subset of PPIs, such that restricting them to values they have in the given satisfying assignment while keeping other PPIs unconstrained, implies that the property fails. In other words, if it is assumed that these PPIs have these values and the property output is set to true, the resulting SAT problem is UNSAT. Below, such minimal subset of PPIs is called a *sensitizing subset* (SS).

To describe the proposed method for abstraction refinement, it is assumed that an integer number called *priority* is assigned to each PPI. This number tells how desirable it is to include a PPI into the SS. In other words, the assignment of priorities to the PPIs imposes an ordering on them, such that a PPI is only included into the SS if an SS with only PPIs of higher priority does not exist.

Note that the SS is not unique. The notion of priority helps us prioritize the PPIs that are included in the subset.

The combinational circuit used for abstraction refinement is a fixed-length unrolling of the abstracted model. In the context of VTA, each time-frame of the unrolling contains only gates included in VTA. The RPIs of the combinational circuit are the PIs corresponding to real PIs of the sequential miter. The PPIs of the combinational circuit are those PIs corresponding to the excluded gates of the abstraction that are fanins of the included gates.

The rules for assigning priority to the PPIs are:

- The constant node, initialized flops of the starting time-frame, and the RPIs get the highest priority. This is because we try to use them whenever possible to imply the property output to fail.
- The PPIs are divided into two categories: (a) those corresponding to abstracted gates included in some time-frame of the VTA; (b) all other PPIs. A higher priority is assigned to PPIs of type (a) and followed by those of type (b). This is done using the rule below. As a result, we control the growth of VTA by adding gates already included in some time frame of the abstraction. Note that the above grouping is used to define two priority ranges. Gates in the first range have priority higher than those in the second range, but we still have freedom to assign relative priority within each range.
- A higher priority for inclusion in the SS is given to those PPIs, which have shorter distance from the PO. (The distance is defined as the number of AIG nodes on the shortest path from a PPI to the PO.) This is motivated by the fact that it is often easier to block a counter-example by adding logic closer to the PO.

Assuming that the priority of PPIs has been assigned, the construction of a SS of PPIs is performed in two traversals of the combinational circuit whose gates have a value assigned according to the given satisfying assignment.

In the *first traversal*, priority is propagated from PIs to the PO in a topological order. When considering AND nodes, the following rules are used to determine priority:

- If both fanins have value 1, the node's priority is the minimum of the priorities of its fanins.
- If both fanins have value 0, the node's priority is the maximum of the priorities of its fanins.
- If fanins have different values, the node's priority is equal to the priority of the node fanin whose value is 0.

At the end of this traversal, each gate of the circuit has priority assigned. This priority is equal to the lowest priority of a PPI needed to produce the listed value at the given gate. Incidentally, if the priority of the PO becomes equal to the higher priority assigned to the terminal gates (constants, flop outputs of the starting frames, or RPIs), the PO failure can be produced without resorting to PPIs, which means a true counter-example has been found.

In the *second traversal*, the SS is determined by traversing the circuit from the PO to the PIs in a reverse topological order. When considering AND nodes, the following rules are used:

- If both fanins have value 1, both of them are traversed.
- If both fanins have value 0, only the fanin with a higher priority is traversed.
- If fanins have different values, only the fanin with value 0 is traversed.
- If a terminal node (a PI, a constant, or a flop output of the first frame) is reached, the check is performed if the

terminal node is a PPI, and if so, it is added to the SS under construction.

The motivation for this is to follow the paths with the highest priority forcing the PO to "fail", and to collect the PPIs forming the required SS at the end of each path.

This method works well in practice. It finds good subsets of PPIs useful to refine the VTA, but it is somewhat expensive in terms of runtime, taking 20-50% of the total time of the abstraction engine, depending on a benchmark.

### 3.4 SAT solver rollback

This subsection describes a new feature added to the SAT solver in ABC to enable efficient implementation of VTA presented in this paper.

The SAT solver in ABC is derived from the original C-version of MiniSAT-1.14 produced by Niklas Sörensson "for those who despise C++" [11]. Since its inclusion in ABC, this version of MiniSAT sustained numerous changes and upgrades, including but not limited to:

- Back-porting of the novel features of MiniSAT 2.0 (e.g. Luby restarts and variable polarity recording).
- Adding the *analyze_final* method used in PDR [12].
- Replacing floating-point-based by integer-based variable activity counting to produce bit-accurate results on different platforms.
- Implementing in-memory proof logging complete with a support for UNSAT core computation, interpolation, garbage collection, and proof consistency checking.
- Upgrading the clause database to use memory pages.

The most recently added SAT solver feature motivated by VTA is called *rollback*. This feature allows the solver to *bookmark* a state of the clause database. Variables and clauses added by the user or derived by incremental SAT runs can be subsequently removed from the solver by invoking the rollback procedure. No attempt is currently made to retain the learned clauses, even if they do not depend on the clauses added since the last bookmark.

The rollback mechanism is useful for performing an exploration of the problem space, by bookmarking a state of the solver, adding more clauses, learning something in the process, and later returning to the bookmarked point and proceeding in a desirable direction.

In the context of VTA, bookmarking is used to remember the state of the solver before the exploration of a new frame begins. When the exploration is over, possibly after several refinement iterations, a new UNSAT core is derived. At this point, the rollback feature can be used to remove useless logic accumulated in the SAT solver during the refinement. After the rollback, only the last UNSAT core is added to the solver. This guarantees that the unrolling is UNSAT in the last frame while keeping the SAT instance compact for the future computation.

### 3.5 Other implementation details

This section lists several details related to the implementation of VTA in ABC as command *&vta*.

The actual unrolling of the miter is not performed. CNF is derived directly using a hash-table, which maps the APs into the corresponding SAT variables. This hash table uses 16 bytes per SAT variable. It can be seen as an unrolling with partial circuit information. Memory usage, however, is dominated by in-memory proof-logging. To reduce the memory usage about 2x, command *&vta* can be tried with switch '-t' to disable the use of "lazy constraints" [14], enabled by default.

Out of the box, command *&vta* runs without resource limits, except for a limit on the abstraction size (-R <num>). With this resource limit defined, *&vta* stops when less than the given percentage (by default, 10%) of internal gates (flops and AND-nodes) is abstracted away. Different ways to impose resource limits on *&vta* could be using runtime (-T <num>), the max number of frames covered (-F <num>) or the max conflicts per SAT call (-C <num>).

To use the abstraction derived by *&vta*, it has to be translated into a proper gate-level abstraction (GLA), similar to that produced by *&gla_cba/&gla_pba*. This is achieved by running *&vta_gla*, followed by *&gla_derive*, followed by, say, a call to PDR (*&put; pdr -v*).

In the verbose mode (*&vta –v*), the following is printed:

Column 1: The number of a time-frame.

Column 2: The size of the GLA, which includes all gates present in any time-frame of the VTA (including all time-frames covered so far).

Column 3: The ratio of nodes in the VTA compared to the unrolled GLA (when it is unrolled for the same depth).

Column 4: The number of conflicts in this frame.

Column 5: The number of CEX during abstraction refinement in this frame.

Column 6: The size (the total number of gates) in the UNSAT core when this frame is proved UNSAT.

The next few columns show the distribution of the number of gates per frame for each UNSAT core.

The final column lists the total runtime since the beginning of the computation.

In summary, the command line may look as follows:

*&r <file>.aig; &ps; &vta -v -T 100; v2p*

where "v2p" stands for "solving VTA using PDR", defined in the file 'abc.rc' as follows:

*alias v2p "&vta_gla; &ps; &gla_derive; &put; pdr -v"*

Command *&ps* here prints the ratios of flops/nodes included in the final GLA. The result of abstraction is passed to PDR [12] for solving in the verbose mode.

### 3.6 Comparison with previous work

Figure 3.6 shows conceptual differences of VTA compared to two main trends of previous work.

The complete bounded unrolling of the sequential miter is shown in Figure 3.6(a). Abstraction based on UNSAT cores extracted from this unrolling is performed in [13].

The gate-level abstraction (GLA) similar to [3] is illustrated in Figure 3.6(b). In this case, only the gates included in the abstraction are replicated in each time-frame. Other gates are not included.

Finally, the proposed approach is shown in Figure 3.6(c). In this case, only a subset of gates needed for gate-level abstraction is included in each time-frame, resulting in a "variable time-frame abstraction" (VTA).

The illustration shows that among the three, VTA leads to the smallest SAT instances because only selected gates need to be included in each time-frame.

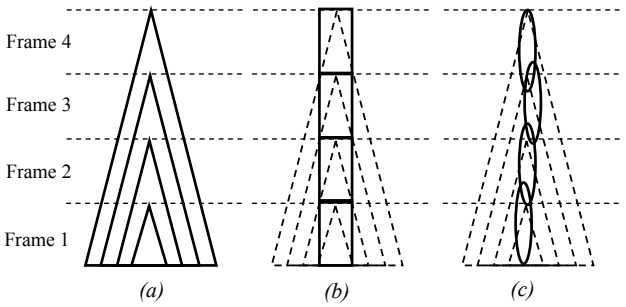


**Figure 3.6: Comparison with previous work: (a) BMC-based abstraction [13], (b) fixed time-frame gate-level abstraction [3], (c) VTA.**

# 4. Experimental results

The proposed abstraction is implemented in ABC [5] as command *&vta.* It was compared against two configurations of the abstraction engine in SixthSense [4], developed at IBM. Using the taxonomy proposed in the introduction, these configurations can be characterized as follows:

- *Config2***:** automatic, SAT-based, counter-example-based, gate-level, fixed time-frame.
- *Config5*: automatic, SAT-based, hybrid, gate-level, fixed time-frame.

The suite of IBM benchmarks submitted to the 2011 Hardware Model Checking Competition [8] is used in this experiment. The results are listed in Tables 1 and 2.

Table 1 compares ABC against SixthSense using flop/gate count, for a fixed BMC depth listed in Column 2. The results show that Config5 produces smaller abstractions than &vta (the abstractions are 20% smaller in terms of gates and 16% smaller in terms of flops).

Table 2 compares ABC against SixthSense in terms of the BMC depth achieved in 5 minutes. The results show that ABC wins in approximately 25% of cases. Comparison against Config2 show that simple counterexample-based abstraction is in many cases much faster and goes deeper in BMC than other localization approaches that may have advantage in terms of abstraction sizes.

Dashes in the tables indicate abnormal termination.

Another experiment, not listed in the tables, has shown that hybrid the SAT-based hybrid flop-based abstraction engine [10] outperforms the proposed method in both runtime and the quality of the resulting abstracted models.

# 5. Conclusions and future work

The paper presents a new method for SAT-based hybrid (both proof-based and counter-example-based) gate-level variable time-frame abstraction. Although the proposed approach is state-of-the-art in both methodology and implementation details, experiments show that, in its current form, it cannot compete with other methods.

Future work may include:

- Using coarser objects to abstract, refine, and derive CNF. Primitives currently used for abstraction and CNF generation are two-input AND nodes and flops.
- Adopting min-cut heuristics [3] to decide what gates to add to the abstraction. This may reduce the number of counter-examples generated and speed up computation.
- Performing the initialized unrolling *before* the method is applied as shown in [2]. For this, proof-logging can be extended to constant propagation and structural hashing, which may improve scalability of the method.

## Acknowledgements

This work was partly supported by SRC contract 1875.001 and NSA grant "Enhanced equivalence checking in crypto-analytic applications". We also thank industrial sponsors of BVSRC: Altera, Atrenta, Cadence, Calypto, IBM, Intel, Jasper, Microsemi, Real Intent, Synopsys, Tabula, and Verific for their continued support.

# 6. References

[1] N. Amla and K. McMillan. "A hybrid of counterexample-based and proof-based abstraction", *Proc. FMCAD'04*, pp. 181-188.

[2] R. Armoni, L. Fix, R. Fraer, T. Heyman, M. Y. Vardi, Y. Vizel, and Y. Zbar, "Deeper bound in BMC by combining constant propagation and abstraction". *Proc. ASP-DAC'07,* pp. 304-309.

[3] J. Baumgartner and H. Mony, "Maximal input reduction of sequential netlists via synergistic reparameterization and localization strategies". *Proc. CHARME'05*, pp. 222-237.

[4] 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.

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

[6] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. "Symbolic model checking without BDDs". *Proc. TACAS'99*, pp. 193-207.

[7] A. Biere, *AIGER format*. http://fmv.jku.at/aiger/

[8] A. Biere, Hardware Model Checking Competition 2011. http://fmv.jku.at/hwmcc11/

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

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

[11] N. Een and N. Sörensson. *MiniSAT*. http://minisat.se/MiniSat.html

[12] N. Een, A. Mishchenko and R. Brayton, "Efficient implementation of property-directed reachability", *Proc. FMCAD'11*.

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

[14] A. Gupta, M. K. Ganai, and P. Ashar, "Lazy constraints and SAT heuristics for proof-based abstraction". *Proc. VLSI'05*, pp. 183-188.

[15] K. McMillan and N. Amla. "Automatic abstraction without counterexamples". *Proc. TACAS'03*, pp. 2-17.

[16] D. Wang, P.-H. Ho, J. Long, J. H. Kukula, Y. Zhu, H.-K. Tony Ma, R. F. Damiano, "Formal property verification by abstraction refinement with formal, simulation and hybrid engines". *Proc. DAC'01*, pp. 35-40.

Table 1: Comparing abstraction size in terms of gate and flop count.

| HWMCC Example | BMC Depth | ABC Flop | ABC Gate | Config2 Flop | Config2 Gate | Config5 Flop | Config5 Gate | ABC/C2 Flop | ABC/C5 Flop | ABC/C2 Gate | ABC/C5 Gate |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 6s0 | 17 | 131 | 2670 | 153 | 3189 | 137 | 3502 | -16.79 | -4.58 | -19.43 | -31.16 |
| 6s1 | 8 | 118 | 1479 | 217 | 2034 | 112 | 1471 | -83.89 | 5.08 | -37.52 | 0.54 |
| 6s2 | 25 | 257 | 2743 | 481 | 6556 | 142 | 1835 | -87.15 | 44.74 | -139.00 | 33.10 |
| 6s3 | 92 | 62 | 856 | 66 | 2079 | 55 | 1802 | -6.45 | 11.29 | -142.87 | -110.51 |
| 6s4 | 177 | 123 | 1641 | 132 | 1636 | 69 | 553 | -7.31 | 43.90 | 0.30 | 66.30 |
| 6s5 | 5 | 559 | 7523 | 995 | 10880 | 362 | 6758 | -77.99 | 35.24 | -44.62 | 10.16 |
| 6s6 | 13 | 201 | 3545 | 235 | 3791 | 201 | 3751 | -16.91 | 0 | -6.93 | -5.81 |
| 6s7 | 20 | 283 | 1425 | 494 | 2424 | 187 | 848 | -74.55 | 33.92 | -70.10 | 40.49 |
| 6s8 | 18 | 227 | 1778 | 378 | 2834 | 239 | 2167 | -66.51 | -5.28 | -59.39 | -21.87 |
| 6s9 | 9 | 290 | 7523 | 215 | 2951 | 197 | 2993 | 25.86 | 32.06 | 60.77 | 60.21 |
| 6s10 | 11 | 558 | 11282 | 415 | 9771 | 186 | 3430 | 25.62 | 66.66 | 13.39 | 69.59 |
| 6s11 | 10 | 528 | 10333 | 385 | 8871 | 158 | 2938 | 27.08 | 70.07 | 14.14 | 71.56 |
| 6s12 | 8 | 507 | 9552 | 414 | 10267 | 289 | 5784 | 18.34 | 42.99 | -7.48 | 39.44 |
| 6s13 | 3 | - | - | 230 | 7573 | 119 | 4602 | 0 | 0 | 0 | 0 |
| 6s14 | 5 | 403 | 9442 | 349 | 10875 | 232 | 7249 | 13.39 | 42.43 | -15.17 | 23.22 |
| 6s15 | 5 | 403 | 9442 | 349 | 10875 | 232 | 7249 | 13.39 | 42.43 | -15.17 | 23.22 |
| 6s16 | 8 | 483 | 8978 | 396 | 9168 | 280 | 6662 | 18.01 | 42.02 | -2.11 | 25.79 |
| 6s17 | 4 | - | - | 180 | 7255 | 87 | 3866 | 0 | 0 | 0 | 0 |
| 6s18 | 3 | - | - | 244 | 7805 | 121 | 4562 | 0 | 0 | 0 | 0 |
| 6s19 | 8 | 149 | 2337 | 267 | 3904 | 186 | 2865 | -79.19 | -24.83 | -67.05 | -22.59 |
| 6s20 | 6 | 177 | 15344 | 190 | 15491 | 169 | 11687 | -7.34 | 4.51 | -0.95 | 23.83 |
| 6s21 | 57 | 166 | 715 | 1377 | 6202 | 168 | 761 | -729.51 | -1.20 | -767.41 | -6.43 |
| 6s22 | 17 | 545 | 6148 | 511 | 5509 | 263 | 3424 | 6.23 | 51.74 | 10.39 | 44.30 |
| 6s23 | 15 | 56 | 644 | 105 | 1221 | 43 | 606 | -87.50 | 23.21 | -89.59 | 5.90 |
| 6s24 | 16 | 255 | 1141 | 781 | 6253 | 229 | 1167 | -206.27 | 10.19 | -448.02 | -2.27 |
| 6s25 | 23 | 23 | 7 | 689 | 2571 | 23 | 7 | -2895.65 | 0 | -36628.60 | 0 |
| 6s26 | 87 | 291 | 1472 | 1105 | 3703 | 246 | 950 | -279.72 | 15.46 | -151.56 | 35.46 |
| 6s27 | 43 | 272 | 1269 | 180 | 696 | 138 | 529 | 33.82 | 49.26 | 45.15 | 58.31 |
| 6s28 | 128 | 46 | 114 | 87 | 221 | 16 | 72 | -89.13 | 65.21 | -93.85 | 36.84 |
| 6s29 | 34 | 692 | 2580 | 975 | 3875 | 442 | 1755 | -40.89 | 36.12 | -50.19 | 31.97 |
| 6s30 | 43 | 140 | 688 | 929 | 6426 | 143 | 716 | -563.57 | -2.14 | -834.01 | -4.06 |
| 6s31 | 31 | 26 | 140 | 91 | 446 | 13 | 107 | -250.00 | 50.00 | -218.57 | 23.57 |
| 6s32 | 497 | 17 | 77 | 15 | 64 | 11 | 56 | 0 | 0 | 16.88 | 27.27 |
| 6s33 | 20 | 122 | 811 | 126 | 850 | 97 | 677 | -3.27 | 20.49 | -4.80 | 16.52 |
| 6s34 | 24 | 89 | 359 | 190 | 753 | 81 | 329 | -113.48 | 8.98 | -109.74 | 8.35 |
| 6s35 | 26 | 133 | 553 | 248 | 960 | 122 | 518 | -86.46 | 8.27 | -73.59 | 6.32 |
| 6s36 | 25 | 104 | 669 | 148 | 904 | 82 | 381 | -42.30 | 21.15 | -35.12 | 43.04 |
| 6s37 | 47 | 217 | 824 | 442 | 2037 | 175 | 687 | -103.68 | 19.35 | -147.20 | 16.62 |
| 6s38 | 7 | 950 | 4569 | 1401 | 7057 | 769 | 4487 | -47.47 | 19.05 | -54.45 | 1.79 |
| 6s39 | 39 | 480 | 3221 | 652 | 6136 | 459 | 3482 | -35.83 | 4.37 | -90.49 | -8.10 |
| 6s40p0 | 36 | 431 | 1585 | 814 | 3077 | 429 | 1647 | -88.86 | 0.46 | -94.13 | -3.91 |
| 6s40p1 | 0 | - | - | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 6s40p2 | 0 | - | - | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 6s41 | 46 | 416 | 1556 | 337 | 1154 | 199 | 626 | 18.99 | 52.16 | 25.83 | 59.76 |
| 6s42 | 21 | 456 | 2646 | 762 | 4601 | 321 | 2190 | -67.10 | 29.60 | -73.88 | 17.23 |
| 6s43 | 25 | 245 | 1277 | 537 | 2960 | 170 | 832 | -119.18 | 30.61 | -131.79 | 34.84 |
| 6s44 | 21 | 411 | 2476 | 721 | 4176 | 366 | 2373 | -75.42 | 10.94 | -68.65 | 4.15 |
| 6s45 | 99 | 99 | 104 | 296 | 1362 | 99 | 104 | -198.99 | 0 | -1209.62 | 0 |
| 6s46 | 99 | 99 | 104 | 417 | 2253 | 99 | 104 | -321.21 | 0 | -2066.35 | 0 |
| 6s47 | 137 | 34 | 114 | 59 | 288 | 21 | 136 | -73.52 | 38.23 | -152.63 | -19.29 |
| 6s48p0 | 6 | 58 | 510 | 56 | 528 | 58 | 466 | 3.44 | 0 | -3.52 | 8.62 |
| 6s48p1 | 7 | 66 | 603 | 62 | 694 | 60 | 566 | 6.06 | 9.09 | -15.09 | 6.13 |
| 6s49 | 15 | 172 | 962 | 178 | 971 | 163 | 980 | -3.48 | 5.23 | -0.93 | -1.87 |
| 6s50 | 93 | 153 | 522 | 888 | 4066 | 78 | 274 | -480.39 | 49.01 | -678.92 | 47.50 |
| 6s51 | 93 | 200 | 772 | 836 | 3980 | 78 | 272 | -318.00 | 61.00 | -415.54 | 64.76 |
| 6s52 | 282 | 98 | 708 | 39 | 108 | 13 | 58 | 0 | 0 | 84.74 | 91.80 |
| 6s53 | 537 | 18 | 67 | 49 | 132 | 19 | 69 | 0 | 0 | -97.01 | -2.98 |
| 6s54 | 25 | 595 | 5245 | 1006 | 8060 | 622 | 6688 | -69.07 | -4.53 | -53.67 | -27.51 |
| Ratio | | | | | | | | -132.82 | 20.07 | -779.57 | 15.69 |

Table 2: BMC depth achieved after running abstraction algorithms for five minutes.

| Design | ABC | Config2 | Config5 | MAX(B,C,D,E) | Winner |
|---|---|---|---|---|---|
| 6s0 | 21 | 21 | 19 | 21 | Config2 |
| 6s1 | 12 | 11 | 9 | 12 | ABC |
| 6s2 | 36 | 26 | 59 | 59 | Config5 |
| 6s3 | 93 | 106 | 133 | 133 | Config5 |
| 6s4 | 178 | 2000 | 2000 | 2000 | Config2,5 |
| 6s5 | 6 | 6 | 6 | 6 | All |
| 6s6 | - | 48 | 14 | 48 | Config2 |
| 6s7 | 29 | 28 | 25 | 29 | ABC |
| 6s8 | 35 | 27 | 21 | 35 | ABC |
| 6s9 | 31 | 96 | 10 | 96 | Config2 |
| 6s10 | 12 | 14 | 12 | 14 | Config2 |
| 6s11 | 11 | 14 | 12 | 14 | Config2 |
| 6s12 | 9 | 10 | 9 | 10 | Config2 |
| 6s13 | 6 | 5 | 4 | 6 | ABC |
| 6s14 | 7 | 6 | 6 | 7 | ABC |
| 6s15 | 7 | 6 | 6 | 7 | ABC |
| 6s16 | 9 | 10 | 9 | 10 | Config2 |
| 6s17 | 7 | 6 | 5 | 7 | ABC |
| 6s18 | 6 | 5 | 4 | 6 | ABC |
| 6s19 | 37 | 59 | 9 | 59 | Config2 |
| 6s20 | 7 | 7 | 7 | 7 | All |
| 6s21 | 58 | 142 | 66 | 142 | Config2 |
| 6s22 | 22 | 32 | 18 | 32 | Config2 |
| 6s23 | 19 | 16 | 16 | 19 | ABC |
| 6s24 | 22 | 20 | 17 | 22 | ABC |
| 6s25 | 24 | 24 | 24 | 24 | All |
| 6s26 | 89 | 2000 | 88 | 2000 | Config2 |
| 6s27 | 61 | 86 | 46 | 86 | Config2 |
| 6s28 | 130 | 130 | 129 | 130 | ABC/Config2 |
| 6s29 | 44 | 71 | 35 | 71 | Config2 |
| 6s30 | 53 | 44 | 44 | 53 | ABC |
| 6s31 | 38 | 33 | 32 | 38 | ABC |
| 6s32 | 498 | 976 | 1024 | 1024 | Config5 |
| 6s33 | 26 | 23 | 21 | 26 | ABC |
| 6s34 | 28 | 61 | 25 | 61 | Config2 |
| 6s35 | 33 | 71 | 27 | 71 | Config2 |
| 6s36 | 28 | 59 | 26 | 59 | Config2 |
| 6s37 | 67 | 48 | 69 | 69 | Config5 |
| 6s38 | 9 | 17 | 8 | 17 | Config2 |
| 6s39 | 40 | 80 | 48 | 80 | Config2 |
| 6s40p0 | 37 | 37 | 37 | 37 | All |
| 6s40p1 | 1 | 1 | 1 | 1 | All |
| 6s40p2 | 1 | 1 | 1 | 1 | All |
| 6s41 | 61 | 63 | 49 | 63 | Config2 |
| 6s42 | 24 | 27 | 23 | 27 | Config5 |
| 6s43 | 57 | 48 | 26 | 57 | ABC |
| 6s44 | 24 | 28 | 23 | 28 | Config2 |
| 6s45 | 100 | 100 | 100 | 100 | All |
| 6s46 | 100 | 100 | 100 | 100 | All |
| 6s47 | 807 | 236 | 149 | 807 | ABC |
| 6s48p0 | 14 | 10 | 7 | 14 | ABC |
| 6s48p1 | 18 | 13 | 8 | 18 | ABC |
| 6s49 | - | 16 | 16 | 17 | Config2,5 |
| 6s50 | 94 | 96 | 201 | 201 | Config5 |
| 6s51 | 96 | 94 | 181 | 181 | Config5 |
| 6s52 | 283 | 539 | 532 | 539 | Config2 |
| 6s53 | 538 | 538 | 538 | 538 | All |
| 6s54 | 26 | 50 | 32 | 50 | Config2 |