GLA: Gate-Level Abstraction Revisited

Alan Mishchenko Niklas Een Robert Brayton Jason Baumgartner Hari Mony Pradeep Nalla

Department of EECS, University of California, Berkeley IBM Systems and Technology Group

{alanmi, brayton}@eecs.berkeley.edu niklas@een.se {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 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 cut-points. In this paper, we propose several improvements to substantially increase the scalability of automated abstraction. In particular, we show how a better integration between the BMC engine and the SAT solver is achieved, resulting in a new hybrid abstraction engine, that is faster and uses less memory. This engine speeds up computation by constant propagation and circuit-based structural hashing while collecting UNSAT cores for the intermediate proofs in terms of a subset of the original variables. Experimental results show improvements in the abstraction depth and size.

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. counterexample-based vs. hybrid
- Flop-level vs. gate-level

The first criterion asks whether abstraction is performed automatically by a tool or manually by a verification engineer. The chosen abstraction can be based on different computation methods; SAT-based techniques have largely replaced BDD-based and other methods. In SAT-based methods, abstraction refinement can rely on computing a sequence of UNSAT cores [16] or counterexamples [17] or by applying a hybrid approach, which utilizes both proofs and counterexamples [1][10]. Finally, in terms of the granularity, abstraction can be flop-level [17][10] or gate-level [3], depending on whether it is constructed using entire next-state logic cones or single gates as primitives.

Using the above taxonomy of abstraction methods, the method presented in this paper can be classified as automatic, SAT-based, hybrid, and gate-level.

The main contribution of this work is in extending the previous hybrid (counterexample-based and proof-based) abstraction [10] in a way that substantially improves scalability of the abstraction engine.

Other contributions include: (a) a new way of computing partial UNSAT cores, resulting in 100x reduction in memory, compared to previous work based on complete proof-logging; (b) a new abstraction refinement framework, which reduces the number of refinement iterations; and (c) a new feature called *rollback*, which allows the SAT solver to return to a previously marked point in solving a problem.

To understand the importance of the main contribution, we start by introducing hybrid abstraction (HA) presented in [10] and outline the two key insights that led to the gate-level abstraction (GLA) described in the current paper.

The implementation of HA constructs a localization abstraction by iteratively computing (1) counterexamples produced while the BMC engine is working on a time frame, and (2) UNSAT cores generated when the SAT solver proves the abstracted model to be UNSAT up to this time frame. In the latter case, it is said that the SAT solver proved the abstraction to be *precise up to this time frame*.

Both counterexamples and proofs are equally important for the efficiency of HA. In particular, the counterexamples indicate locations in the circuit structure where abstraction is not adequately precise. This directs the refinement procedure to aggressively add more logic to the abstracted model. When eventually abstraction becomes precise up to a given time frame, an UNSAT core returned by the SAT solver is used to prune superfluous logic, resulting in a smaller abstraction that is still adequately precise. A major performance drawback of HA is that the BMC procedure [6] producing counterexamples cannot leverage circuit-based simplifications (such as constant propagation and structural hashing) without jeopardizing the proof-logging required by the SAT solver to generate UNSAT cores. Another important drawback of HA is that it relies on the complete proof whose memory requirements run into gigabytes after several minutes of computation. This makes it necessary to store the proofs on disk, which slows down computation and make implementation cumbersome.

The key insight to overcome these limitations is the following: When a precise abstraction up to a given time frame is found, and the abstraction engine is about to start working on the next time frame, we can choose to always keep the gates added to the abstraction so far, that is, never remove them while possibly adding some new gates to make the abstraction more precise in future time frames. Thus when we analyze the UNSAT core at the end of a time frame we always keep all the gates used at the end of previous timeframes.

Since we committed to keep these gates in the abstraction until the end of the run of the abstraction engine, we can allow circuit-based simplification to be applied to them. For example, the constant initial state can propagate and structural hashing can reduce the part of the circuit

978-3-9815370-0-0/DATE13/©2013 EDAA

included in the abstraction. This simplification is crucial for the scalability of the SAT solver working on multiple time frames during BMC. Meanwhile, UNSAT cores can be recorded in terms of only gates added to the abstraction in a particular time frame, since the previously added gates are never removed. Once refinement is completed, a new set of gates is added to the abstraction and kept till the end.

Another key insight allows a dramatic reduction in memory usage for proof-logging: After completing a time frame and performing the refinement, we commit to those gates that have been added to the abstraction so far. As a result, future UNSAT cores, if any, can be expressed in terms of only new gates to be added in the next time frame. This allows recording incremental UNSAT proofs and generating cores in terms of relatively few SAT variables.

Moreover, there is no need to store the structure of the UNSAT proof as done by a typical proof-logger. It is enough to associate each clause with a bit-string encoding new SAT variables using positional notation. This insight leads to a 100x reduction in memory, compared to using full proofs recorded by state-of-the-art proof-loggers, such as the one in MiniSAT. In practice, this means going from 1GB to 10MB on a typical model checking instance that takes about 5 minutes to compute a precise abstraction.

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, sequential logic networks, And-Inverter Graphs (AIGs), time frame unfolding, bounded model checking (BMC), SAT solving, counterexample- and proof-based abstraction, etc., are assumed to be well known. Only definitions specific to this paper are given 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 is 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 unfolding 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 unfolding. For scalability of subsequent verification, this subset of gates should be as small as possible. Although given the heuristic nature of the abstraction-refinement process, generally the resulting abstraction may include some gates which are unnecessary.

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

An *included (excluded) gate* is a gate included (not included) in the abstracted model. In the present work, the constant node and the property output are always included.

An *incremental UNSAT core* in frame K is defined as follows: Given is (a) a sequential miter, (b) an abstraction, which is precise up to depth K-1 (K > 0), and (c) a set S of AIG gates not included in the abstraction. For set S to be an incremental UNSAT core in frame K, the SAT instance containing the following constraints should be unsatisfiable:

- all the gates included in the abstraction in all K frames;
- all the gates belonging to S in all K frames;
- constraints for the initial state in the starting frame;
- the property output fails in frame K.

In this paper, *GLA* stands for the proposed method called "gate-level abstraction" and for an abstraction produced by this method. In this sense, a GLA of depth K is a set of AIG gate IDs, which should be included in each of the first K time frames of the unfolding of the abstracted model to guarantee that the property output does not fail in any of the time-frames from 0 to K-1, inclusive.

3. Algorithm

3.1 The main idea

The proposed method is motivated by the need to improve scalability of abstraction. It differs from other methods in the way it handles gates included in the abstraction while refining the abstraction in the latest time frame.

The gates included in the abstraction *before* frame K can be constant-propagated and structurally-hashed to reduce the size of the SAT problem. When the refinement in frame K is finished, an incremental UNSAT core can be computed only in terms of the new gates. Once this UNSAT core is available, the set of new gates belonging to the core can be added to the abstraction, resulting in the abstraction of depth K. From now on, all the gates included in this abstraction will be considered "old", while some "new" gates may be added in time frame K+1 if refinement of the abstraction is required in this frame.

The abstraction refinement process terminates upon some configurable scenario: e.g., when a specified number of frames is explored, when a resource limit is reached, or when there was no refinement in several recent timeframes. If a true counterexample is discovered at any time, computation terminates, and the counterexample is returned. If the SAT solver exceeds the allotted resources, the current abstraction is returned.

3.2 Abstraction framework

The abstraction framework is described in Figure 3.2.

3.3 Priority-based abstraction refinement

This subsection describes our priority-based abstraction refinement (PBAR), which is a special case of counterexample-based abstraction refinement. PBAR is orthogonal to GLA and compatible with other abstraction methods. The pseudo-code is in Figure 3.3.

```
aig deriveAbstraction (
   aig A,          // A is a sequential miter
   params P )      // P is a set of parameters
{
   if ( aig A comes without an initial abstraction )
      initialize the abstraction of A to contain the output gate;
   initialize time frame counter f to 0 or to a 'starting time frame';
   while (resource limits are not exceeded)  {
      unfold the current abstraction in time frame f;
      simplify gates added to the abstraction before the last restart;
      do not simplify gates added after the last restart;
      status = callSAT( unfolded abstraction, resource limits );
      if ( status == UNDEF ) return "current abstraction";
      if ( status == SAT ) {
         // perform refinement
         bookmark the current state of the SAT solver;
         while ( status == SAT ) {
            perform priority-based abstraction refinement;
            if ( refinement is impossible )
                 return "true counterexample";
            add new gates to all frames without simplification;
            status = callSAT(unfolded abstraction, resource limits);
            if ( status == UNDEF ) return "current abstraction";
         }
      }
      // the status of the previous SAT call should be UNSAT
      assert( status == UNSAT );
      if ( abstraction refinement took place ) {
         compute UNSAT core in terms of gates added in frame f;
         rollback SAT solver to the previous bookmark;
         add simplified logic of the UNSAT core to all time frames;
      }
      if ( ratio of gates added to the abstraction since the last restart
              exceeds a predefined limited )   {
         restart the framework;
         add simplified logic of unfolded time frames up to frame f;
      }
      f = f + 1;
   }
   // abstraction finished a predefined number of time frames
   return "current abstraction";
}
```

Figure 3.2. Abstraction framework.

Consider the abstracted model, A^M, and its sequential unfolding for as many time frames as needed to get the property output to fail in the last time frame according to the provided counterexample C. This unfolding is a combinational circuit with a single primary output (PO), which represents the output of the original miter in the last time frame, and two types of primary inputs (PIs), called *real PIs* (RPIs) and *pseudo PIs* (PPI), depending on whether they are PIs of the original miter or intermediate cut-points produced by the abstraction engine. Consider also a complete assignment of the PIs, given by the counterexample, which forces the PO to fail.

The refinement algorithm produces a minimal subset of PPIs, such that restricting them to values in the given CEX 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 does not fail, the resulting SAT problem is UNSAT. If the resulting subset of PPIs is empty, the property failure does not depend on values at the PPIs. It means that the given CEX is a true counterexample to the original verification problem and thus cannot be ruled out by any refinement.

```
logic gates performPriorityBasedAbstractionRefinement (
   aig A,              // A is a sequential miter
   abstraction M,      // M is a mapping of objects of A into {0, 1}
   cex C  )            // C is a counterexample for abstraction A^M
{
   traverse the combinational unfolding of the abstracted model
        from PIs to PO and assign priorities to intermediate nodes;
   traverse the combinational unfolding of the abstracted model
        from PO to PIs and select a justisfying subset (JS) of PPIs;
   reduce the JS by removing those PPIs that when added
        do not create recovergent paths in the abstraction model;
   return logic gates driving the reduced set of PPIs;
}
```

Figure 3.3. Priority-based abstraction refinement.

Below, such a minimal subset of PPIs is called a *justifying subset* (JS) of the refinement problem. Note that the JS is not unique. Several JS's of different cardinality can exist for the same refinement problem. However, as our experiments have shown, the number of different JS's in realistic refinement problems is typically not large.

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

When assigning priorities to the PIs, the constant node, the constants representing initial values of the flops in 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, before including other PPIs into the resulting JS. The PPIs get priority in their natural order of appearance in the topologically traversed AIG of the sequential abstraction model. The PPIs appearing in different time frames get the same priorities.

Assuming that the PPI priority is assigned and each node included in the abstraction is associated in each timeframe with the value it has in the counterexample, a JS of PPIs is constructed in two traversals of the unfolding.

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 counterexample value 1, the node's priority is the minimum of the priorities of its fanins.
- If both fanins have counterexample 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 combinational unfolding has a priority assigned. This priority is equal to the lowest priority of a PPI needed to produce the assigned value at a gate. Incidentally, if the priority of the PO is equal to the higher priority assigned to the terminal gates (constants, flop outputs of the starting

frames, or RPIs), the PO failure is produced without PPIs, which means a true counterexample has been found.

In the *second traversal*, the JS is determined by traversing the circuit from the PO to the PIs in a reverse topological order. The following rules are used at each AND node:

- 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' values differ, only the 0-fanin is traversed.
- If a terminal node (a PI, a constant, or a flop output of the first frame) is reached and the terminal is a PPI, it is added to the JS under construction.
- Immediately after traversing an intermediate node or adding a new PPI to the JS, fanouts of the node or the PPI are traversed and all the nodes whose values are implied by the traversed nodes are labeled as traversed. This allows us to reduce the size of the JS by reusing as many justified intermediate assignments as possible.

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 JS at the end of each path.

This method works well in practice. It finds small subsets of PPIs useful to refine the GLA while its runtime typically does not exceed 5% of that of the abstraction engine.

3.4 SAT solver rollback

This subsection describes a new feature added to the SAT solver in ABC to enable efficient implementation of GLA.

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 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 GLA 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 rolling back to the last bookmark. Only one bookmark is currently supported. No attempt is currently made to retain the newly 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 rolling back to the bookmarked point and proceeding in a more desirable direction.

In the context of GLA, bookmarking is used to remember the state of the solver before exploring a new frame. When the exploration is over, possibly after several iterations of refinement, a new UNSAT core is derived. At this point, the rollback removes the logic accumulated in the SAT solver during the refinement. After the rollback, only gates in the last UNSAT core are added to the SAT solver in all timeframes. This guarantees that the unfolding is UNSAT in the last frame while keeping the SAT instance compact for the future computations.

3.5 Usage details

This section lists important details related to the implementation of GLA in ABC as command *&gla*.

Out of the box, *&gla* runs without resource limits. It can be controlled by limiting: runtime (-T <num>), the max number of frames covered (-F <num>), the max conflicts per SAT call (-C <num>), or the abstraction size (-R <num>). With the latter resource limit, *&gla* stops when less than a given number of gates is abstracted away.

To view the parameters of the abstraction derived by *&gla*, run *&ps*. To derive the abstracted model, run *&gla_derive*, followed by a call to a prover of your choice.

In the verbose mode (*&gla –v*), the following information is printed while abstraction is being performed:

Column 1: The number of a time-frame.
Column 2: Percentage of objects included in the abstraction.
Column 3: The size of the abstraction in terms of gates (logic nodes and flops, excluding primary inputs).
Column 4: The number of pseudo-primary inputs (PPIs).
Column 5: The number of flip flops (FF).
Column 6: The number of logic nodes (LUT).
Column 7: The number of conflicts in this frame (Confl).
Column 8: The number of CEXes in this frame (CEX).
Column 9: The number of variables in the SAT solver (Vars).
Column 10: The number of problem clauses (Clas).
Column 11: The number of learned clauses (Lrns).
Column 12: Runtime since the beginning (Time).
Column 13: Memory used by SAT solver and proof (Mem).

In summary, the command line to run GLA may look as follows: *&r <file>.aig; &gla -v -T 100; &gla_derive; &ps; &put; pdr,* where *&gla_derive* derives the abstracted model and command *pdr* tries to prove the abstracted model UNSAT using PDR [12]. Command *&ps* prints the ratios of flops/nodes included in the resulting abstraction. Alternatively, command line *&r <file>.aig; &gla -vq* runs abstraction and PDR for the abstracted model concurrently.

Two granularities of CNF construction are supported in *&gla* (with switch *–c*): logic nodes with up to 5 fanins [12] (default) and 2-input AND nodes. In practice, the former is better because the CNF is more compact (SAT solving is faster) and the granularity of abstraction is increased (resulting in fewer refinement iterations).

3.6 Comparison with previous work

The main difference of GLA compared to previous work on hybrid abstraction [3][10], is the ability to simplify the abstracted part of the circuit and to perform UNSAT core computation incrementally. Minor differences are:

reference [3] uses min-cut heuristics while we found them not useful when relying upon PDR [12] as a subsequent proof technique; reference [10] uses flop-based abstraction while we found that a more fine-grain GLA scales better.

A different way of improving the UNSAT core computation during abstraction was developed in [2].

4. Experimental results

The proposed approach to abstraction (GLA) is implemented in ABC [5] as command *&gla*. GLA is compared against hybrid flop-based abstraction (ABS) [10] and two versions of GLA: *GLAnp* with UNSAT core based on full proof and *GLAn* based on incremental UNSAT core. Unlike GLA, both disable constant propagation and structural hashing across the time frames.

The suite of IBM benchmarks from [8] (except two easy SAT instances *6s40p1* and *6s40p2* are removed) was used in this experiment. The experiment used one core of an Intel Xeon CPU E5-2670 2.60GHz. Learned clause removal, abstraction manager restarts, and early termination, are disabled to reduce "noise", resulting in the following call: *&gla* [*-n*] [*-p*] *–L 0 –P 0 –R 0 –T 300.*

Table 1 shows the initial statistics (name, primary input count, flop count, and AND gate count) and compares the four abstraction engines (ABS, GLAnp, GLAn, and GLA) in terms of the following metrics:

- Depth (abstraction depth achieved in 300 seconds)
- AND (the number of ANDs in the abstracted model)
- Mem (the amount of RAM used for the UNSAT core computation, in megabytes)

At the bottom, "Geom" shows geometric averages of the corresponding columns and "GeomFF" shows geometric averages of flop counts in the abstracted model. (The actual data are omitted due to page limits.) Dashes in the table indicate that the tool core-dumped.

The results lead to the following observations:

- GLA finds abstractions that are 59% deeper than those found by ABS and 10% deeper than (GLAn),
- GLA produces abstractions that are close to ABS in terms of flops but 36% smaller in terms of AND gates,
- GLA and GLAn use on average 500x and 160x less memory for UNSAT cores than GLAnp, which computes a full proof rather than an incremental proof.

5. Conclusions and future work

The paper presents a revised approach for SAT-based hybrid gate-level abstraction, which facilitates checking safety properties represented by large sequential miters, when a relatively small abstraction exists. Miters with a few million gates resulting in abstractions with a few thousand gates have been successfully processed by the algorithm.

The method differs from previous work in that it is more scalable (due to constant propagation and structural hashing) and uses less memory (due to incremental UNSAT core computation without complete proof logging).

Experiments show that the improved abstraction engine works well for industrial verification benchmarks, substantially outperforming previous work and comparing favorably against state-of-the-art industrial tools.

Future work may include:

- Improving abstraction refinement by performing a more detailed structural analysis.
- Developing a more scalable way of refining deep failures of abstraction using partial counterexamples.
- Applying a similar approach to make interpolation-based model checking more scalable.

Acknowledgements

This work is supported in part 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, Mentor Graphics, 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 N. Sorensson, "Applying logic synthesis to speedup SAT", *Proc. SAT '07*, pp. 272-286.

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

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

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

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

[17] 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: Experimental evaluation of abstraction engines using IBM HWMCC benchmarks.

Example name	PI Base	FF Base	AND Base	Depth ABS	Depth GLAnp	Depth GLAn	Depth GLA	AND ABS	AND GLAnp	AND GLAn	AND GLA	Mem GLAnp	Mem GLAn	Mem GLA
6s0	207	157	3549	20	25	24	25	3522	2792	2792	2749	281	4	1
6s1	45	291	3023	9	10	10	10	1984	1771	1771	1573	473	2	1
6s2	856	781	11945	51	194	192	347	9152	2647	2647	1652	561	4	1
6s3	156	68	3504	59	1312	1267	1718	3480	792	792	745	849	1	1
6s4	209	202	2451	1075	3005	3368	4978	1720	535	535	477	873	2	1
6s5	141	2519	28985	5	6	6	5	19090	12618	12618	12639	625	115	1
6s6	168	429	4771	19	65	67	118	4743	4550	4550	4550	1601	16	1
6s7	45	504	2530	28	33	34	35	1811	1759	1759	1765	1457	2	1
6s8	86	396	3016	31	35	35	52	2600	2300	2300	2285	737	2	1
6s9	252	607	15555	29	460	481	569	10533	996	996	883	2865	1	1
6s10	244	598	15373	14	13	13	12	15098	10701	10701	10378	1105	115	29
6s11	244	598	15430	14	14	14	12	15156	13127	13155	10037	985	104	95
6s12	245	598	15439	9	10	10	9	15103	10848	11418	10003	809	32	1
6s13	439	811	25083	5	6	6	5	17233	11752	11810	11929	793	58	1
6s14	439	811	24927	6	7	7	6	16916	12981	13030	12345	1249	29	1
6s15	439	811	24927	6	7	7	6	16916	12981	12981	12345	1233	29	1
6s16	249	608	14137	9	10	10	9	12882	10001	10001	9849	1081	43	1
6s17	450	819	22419	6	7	7	6	15422	12299	12299	11817	1401	27	1
6s18	450	819	22559	6	6	6	5	15557	11642	11642	11663	873	22	1
6s19	266	607	14308	18	429	460	494	10296	928	928	835	3033	1	1
6s20	49	201	30251	7	7	7	6	28787	17962	17919	16469	1713	39	1
6s21	155	3795	20098	100	137	134	172	1904	1744	1744	1303	1369	5	1
6s22	73	1126	15983	30	36	36	28	10588	4412	4412	3457	1409	6	1
6s23	12	10009	61603	19	19	19	20	771	686	686	673	641	1	1
6s24	25	1456	10537	19	23	23	20	5695	3938	3938	2174	441	11	1
6s25	131	1718	6615	22	23	23	22	6582	6576	6576	6570	977	1	1
6s26	247	2654	9980	1292	-	177	86	1917	-	1869	3361	-	11	1
6s27	144	2707	10239	50	69	69	57	622	598	598	440	969	1	1
6s28	4	2269	9974	128	129	130	128	3987	3683	3277	2331	633	1	1
6s29	4	2247	9897	55	62	61	51	5107	4500	5046	3307	1897	46	1
6s30	32994	1195	102535	51	61	62	66	1590	810	810	772	1225	7	1
6s31	17	197	1355	29	34	34	40	387	121	121	150	297	1	1
6s32	8	1773	7482	999	2062	2062	2060	62	2104	2104	327	537	1	1
6s33	31	142	954	26	26	26	26	844	810	810	789	825	1	1
6s34	77	1564	9460	39	28	28	59	500	453	453	971	889	1	1
6s35	77	1572	9608	40	33	33	51	1252	868	868	877	265	2	1
6s36	74	1072	7155	26	28	28	26	688	636	636	611	1585	1	1
6s37	42	753	3707	79	-	94	72	3015	-	1922	1461	-	26	1
6s38	343	1931	10787	13	18	18	18	7491	5750	5750	5459	625	7	1
6s39	65	698	6712	70	64	60	63	6498	4348	5406	4544	177	12	1
6s40p0	249	5608	30430	39	36	37	35	23776	13737	13602	8152	41	1	1
6s41	19	959	3274	65	74	74	72	2389	1892	2620	2644	769	4	1
6s42	76	1211	8345	28	30	31	29	4847	4050	4050	5188	713	7	1
6s43	30	965	7408	71	129	133	284	2349	1526	1526	1940	1025	2	1
6s44	76	1211	8635	28	30	30	29	5061	4605	4605	4075	1169	7	4
6s45	91	651	4340	98	99	99	98	3985	3569	3569	3728	1409	1	1
6s46	91	651	4208	98	99	99	98	3944	3566	3566	3622	1209	1	1
6s47	34	815	4101	274	616	624	892	321	334	338	198	1377	1	1
6s48p0	72	66	795	8	11	11	10	764	568	568	549	97	1	1
6s48p1	72	66	791	7	12	12	10	763	558	558	563	121	1	1
6s49	17	180	1020	14	15	15	14	1003	955	955	946	201	1	1
6s50	1570	3107	16700	77	311	317	654	1844	801	801	482	1905	2	1
6s51	1570	3107	16701	85	290	298	421	1758	835	835	460	1657	2	1
6s52	35	208	1228	567	549	549	541	1074	886	886	910	729	2	1
6s53	35	208	1230	536	537	537	536	1187	1179	1179	1156	385	1	1
6s54	144	1660	13411	46	47	45	38	8816	7721	7721	8230	1417	25	1
Geom				1.000	1.486	1.493	1.585	1.000	0.716	0.724	0.638	1.000	0.006	0.002
GeomFF								1.000	1.074	1.082	0.990			