URL: http://www.elsevier.nl/locate/entcs/volume23.html 12 pages

Evaluating Semi-Exhaustive Verification Techniques for Bug Hunting

Ranan Fraer, Gila Kamhi, Limor Fix, Moshe Y. Vardi*

Logic and Validation Technology, Intel Corp., Haifa, Israel
**Dept. of Computer Science, Rice University*

Abstract

We propose a methodology to evaluate a rich set of BDD subsetting heuristics with respect to bug hunting and apply it to a set of real-life Intel designs. Our results illustrate that the evaluation metrics used to rate these heuristics in previous work were not tuned for bug-finding efficiency, which we believe is the major criterion that the heuristics need to meet.

1 Introduction

The increasing complexity of today's cutting-edge microprocessor designs makes the detection of functional bugs a much more challenging task. Such bugs can be very expensive when caught late only at the post-silicon stage; therefore, there is a strong need to guarantee the absence of hardware design errors before manufacture.

Functional RTL validation is addressed today by two complementary technologies. The more traditional one, *simulation*, has high capacity and full chip simulation is possible. However, simulation is rarely exhaustive, covering only a tiny fraction out of all the possible behaviors of the design, and leaving the door open for subtle bugs to slip in.

This has prompted the use of *formal verification* as an alternative validation technique. Formal verification guarantees full coverage of the entire state space of the design, thus providing high confidence in its correctness. Still, the more automated and therefore the more popular formal verification, *symbolic model checking*, has a severe problem of *limited capacity*. State-of-the art model checkers can hardly verify moderate size designs of the order of hundreds of sequential elements.

In recent years, a *hybrid* approach that merges the strengths of the two validation technologies has been introduced [1–5,9]. *Semi-exhaustive verification* addresses the concerns of practicing verifiers by shifting the focus from verification to *falsification*. Rather than ensuring the absence of bugs, it turns

the verification tool into an effective bug hunter. This hybrid approach aims at improving over both simulation and formal verification in terms of state space coverage and capacity respectively.

Semi-exhaustive verification is not only useful when the design is too large to be fully verified against the specification. It can also pay off in the early stages of the verification, where the bugs found are not real design errors, but indicate holes in the specification. In this stage, that we call *specification debugging*, the verifier is in the loop of model checking and specification modification based on the feedback from the symbolic model checker. The turn-around time of the model checker becomes then very critical to the productivity of the verifier. Therefore, the benefit of semi-exhaustive verification is clearly observed at this stage.

Our usage of semi-exhaustive verification follows the lines of [1,3,5,9], being based on subsetting the frontiers during state space exploration, whenever these frontiers reach a given threshold. While previous work focuses on the algorithmic issues, in particular on the BDD subsetting heuristics, the evaluation of the various algorithms is based only on the number of visited states. Although this metric is interesting as a coverage indicator, it does not say anything about the success of this technique as a bug finding tool. Covering more states, and adventuring deeper in the state space is indeed supposed to increase the chances of bug finding, but no experimental data has been provided to back this assumption.

The main contribution of our work is in evaluating a rich set of existing semi-exhaustive algorithms with respect to bug finding. We propose an accurate methodology to evaluate a rich set of BDD subsetting heuristics with respect to bug hunting and apply it to a set of real-life Intel designs. This confirms our intuition that neither the number of visited states, nor the density of the approximation are sufficient criteria for effective bug finding. It also suggests various ways of combining or improving existing algorithms with respect to bug hunting.

The paper is organized as follows. In Section 2, we present an overview of semi-exhaustive verification with respect to exhaustive verification. Section 3 surveys the different semi-exhaustive techniques that we have evaluated with respect to bug finding. The methodology that we deployed to evaluate these techniques is described in Section 4. In Section 5 we interpret the experimental results obtained on a set of real-life Intel designs. Section 6 concludes by suggesting future directions of research.

2 Semi-exhaustive Verification versus Exhaustive Verification

A common verification problem for hardware designs is to determine whether every state reachable from a designated set of initial states lies within a specified set of "good states" (referred to as the *invariant*). This problem is vari-

ously known as *invariant verification*, or *assertion checking*.

Invariant verification can be performed by computing all states reachable from the initial states and checking that they all lie in the invariant. This reduces the invariant verification problem to the one of traversing the state transition graph of the design, where the successors of a state are computed according to the transition relation of the model. Moreover, traversing the state graph in a breadth-first order makes possible to work on sets of states that are symbolically represented as BDDs [6]. This is an instance of the general technique of *symbolic model checking* [7].

According to the direction of the traversal, invariant checking can be based on either *forward* or *backward analysis*. Given an invariant I and an initial set of states *Init*, *forward analysis* starts with the BDD for *Init*, and uses the *Img* operator to iterate up to a fixed-point, which is the set of states reachable from *Init*. Similarly, in *backward analysis*, the *PreImg* operator is iteratively applied to compute all states from which it is possible to reach the complement of the invariant.

The primary limitation of both *exhaustive* forward and backward analysis is that the BDDs encountered at each iteration, commonly referred as *frontiers*, can grow very large leading to a blow-up in memory or to a verification time-out. *Semi-exhaustive* verification addresses this issue by keeping the size of the frontiers under control. More precisely, each time the size of the current frontier reaches a given threshold, only a subset of the frontier is retained to proceed further.

This heuristic can be regarded as a mixture of breadth-first and depth-first search that still enjoys the benefits of the symbolic BDD representation. With the risk of missing some reachable states, hence the name of *semi-exhaustive*, it can allow the verification to cover more states, and detect bugs that would be hard to reach otherwise.

3 BDD Subsetting Algorithms

As expected, the effectiveness of the semi-exhaustive verification is very sensitive to the nature of the algorithm employed for subsetting the frontiers. A number of BDD subsetting algorithms have been proposed lately in the model checking literature. Each of them is necessarily a heuristic, attempting to optimize different criteria of the chosen subset. This section surveys the subsetting heuristics that we have used in our experiments, together with a brief explanation of the underlying algorithms.

An important class of heuristics takes the *density* of the BDDs as the criterion to be optimized, where density is defined as the ratio of states represented by the BDD to the size of the BDD. This relates to the observation that large BDDs are needed for representing sparse sets of states (as it is often the case for the frontiers). Removing a few isolated states can thus lead to significant reductions in the size of the BDDs.

Ravi and Somenzi [1] have introduced the first algorithms for extracting dense BDD subsets, *Heavy-Branch*(HB) and *Short-Path* (SP). *Heavy-Branch* (HB) subsetting counts the size and the number of states represented by each node of the BDD graph. Starting from the root, it discards the lighter branch of each node (in terms of number of states) while subtracting the number of nodes contributed by this branch, until the size threshold is reached.

Short-Path (SP) attempts to preserve the short paths of the BDD, based on the observation that such paths contribute many states to the set for relatively few nodes. It proceeds by computing the length of the shortest path going through each node, evaluating the maximum length of paths that have to be preserved, and then discarding nodes with no short paths going through them.

Independently, Shiple proposed in his thesis [8] the algorithm *Under-Approx* (UA) that also optimizes the subset according to the density criterion [1]. Like HB and SP this algorithm is also based on discarding some nodes of the BDD. However, the selection of the nodes to be discarded is based on a more intricate analysis - it computes for each node a lower bound on the increase in density that would follow from discarding this node. Unlike HB and SP that can occasionally decrease the density of the result, UA can be run in a "safe" mode. An additional parameter, *quality*, is used to define the minimum improvement in density for the replacement of a node to be acceptable.

Recently, Ravi, McMillan, Shiple and Somenzi [9] proposed *Remap-Under-Approx* (RUA) as a combination of UA with more traditional BDD minimization algorithms like *Constrain* and *Restrict* [10]. Rather than simply discarding a node, such algorithms *remap* it to its brother, making their father redundant as well and increasing the sharing of the nodes. Like in UA, the benefit of replacing is computed for each node, but this time for all possible replacement operations, the most profitable one being selected at the end.

A combined algorithm is *Compress* (COM), which applies first SP with the given threshold and then RUA with a threshold of 0 to increase the density of the result. Although more expensive, the combination of the two algorithms is supposed to produce better results.

The *Saturation* (SAT) algorithm [3,5] is based on a different idea. Rather than keeping as *many* states as possible, it attempts to preserve the *interesting* states. In the context of [3,5] the control states are defined as the interesting ones. The heuristic makes sure to *saturate* the subset with respect to the control states, i.e. that each possible assignment to the control variables is represented exactly once in the subset. In terms of BDDs, this is implemented by Lin and Newton's *Cproject* operator [11].

[1] Actually, the original algorithm of Shiple considered a convex function of the number of the states and nodes, but the CUDD implementation that we used optimizes the density.

4 Evaluation of Semi-exhaustive Search Techniques

This section describes the methodology and the criteria that we deployed to evaluate the frontier subsetting heuristics with respect to bug finding efficiency.

4.1 Evaluation Methodology

We selected for the purpose of the evaluation four real-life Intel *invariant* verification tasks that indicate bugs in the model or specification. We gave priority to the verification tasks with long (i.e., at least 20 cycles) counter-example traces to get rich data with respect to frontier subsetting. All verification tasks except for CKT4 have been identified to have specification bugs. In CKT4 we have planted a design bug for the evaluation purposes.

We report three modes of operation relative to the aggressiveness level of the approximation, namely, high, medium and low level, according to the threshold that triggers the subsetting. Highly aggressive approximation corresponds to the setting of the threshold to 50000 BDD nodes; whereas the medium and low levels corresponds to the setting of the threshold to 100000 and 200000 BDD nodes respectively.

As for the frontier subsetting, we evaluated all the techniques described in Section 3 in the following conditions:

- SP, HB and COM have been evaluated as they are, with the *threshold* taking one of the values mentioned above.
- In the case of UA and RUA, the additional parameter *quality* seems to be at least as important as the *threshold*. We have chosen to evaluate each of them with 3 different values for quality:
  - UA and RUA take the default value *quality* = 1, meaning that only replacements that do not decrease the density are acceptable.
  - UA+ and RUA+ are even more conservative in taking *quality* = 1.5, which means that only replacements bringing 50% increase in density are acceptable.
  - Conversely, UA- are RUA- are more relaxed, taking *quality* = 0.5, which means that occasionally they can decrease the density but not more than 50%.
- Unlike [3,5] where SAT is used in conjunction with the control variables, our use of SAT saturates the frontier with respect to the specification variables. More precisely, each temporal property is translated internally into a property checker. It is the variables of this checker that are used for saturation.
- For comparison, we have also included in our data the results produced by fully exhaustive on-the-fly forward model checking, denoted as EXH.

Finally, note that the heuristics were compared with identical settings of the model checker, with the same initial order file and with no dynamic re-ordering

in order to reduce the external parameters that affect the results.

4.2 Evaluation Criteria

The interpretation of our results will be structured in three stages. The first stage reports the summary of our results with respect to bug finding. This reflects the perspective of a casual user, who only has a black-box view of the subsetting heuristics.

In a second stage we try to gain more insight by collecting various statistics on the behavior of the heuristics along the run. Such statistics will include:

- N - the number of approximations.
- $AvgT$ - the average time required for an approximation.
- $AvgSt$ - the average percentage of states conserved by an approximation.
- $AvgNd$ - the average percentage of nodes conserved by an approximation.

The last two quantities reflect the aggressiveness of each heuristic. It is important to consider them separately, rather than looking just at their ratio, i.e. the *density*. One of the main results of our experiments is that density alone is not a sufficient condition for success: heuristics achieving a very good density can often miss the bug due to their high aggressiveness and vice-versa.

The most accurate analysis is performed in the third stage. It assumes that backward analysis is possible on the model at hand, in order to compute the set of *target* states (that could lead to a bug). This computation starts from the *error states* (that violate the invariant) and iteratively applies the *PreImg* operator till reaching a fixed-point.

The set of successive frontiers obtained by approximation during the forward analysis are then graded according to their success in keeping target states and throwing away other states. For this purpose, we define the following grading function which measures the target states covered by the approximated frontier, as a percentage of a total number of target states:

$$\text{Grade}(approxFrontier) = \frac{\text{states}(approxFrontier \cap Target)}{\text{states}(Target)}$$

As a by-product, this evaluation technique illustrates the impact of the *target enlargement* [3,4] technique for bug-finding. This technique aims at enlarging the set of error states (violating the invariant) with some of the closest *target frontiers*. The intuition is that a larger target increases the chances of hitting a bug faster during on-the-fly forward analysis, and consequently reduces the amount of searching that needs to be done. Therefore, techniques that get a high grade are expected to perform better (i.e., to find the bug faster and with less memory consumption) when applied together with *target enlargement*.

Ckt	CKT1, 96 latches, 72 inputs						CKT2, 165 latches, 33 inputs								
thresh	100000 nodes			50000 nodes			200000 nodes			100000 nodes			50000 nodes		
	R	iter	time	R	iter	time	R	iter	time	R	iter	time	R	iter	time
EXH	B	24	638				B	9	79						
SP	B	26	550	B	36	450	B	9	75	B	9	66	B	9	62
HB	D	14	244	D	10	138	B	9	55	B	9	44	B	9	26
UA+	B	26	560	B	26	292	B	9	57	B	9	70	B	9	78
RUA+	B	26	559	B	26	291	B	9	58	B	9	72	B	9	81
UA	B	24	258	D	10	138	B	9	58	B	13	133	B	9	36
RUA	B	24	355	D	10	138	B	9	58	B	13	139	B	9	36
UA-	D	14	244	D	10	138	B	9	58	B	9	51	B	13	70
RUA-	B	150	7616	B	130	3378	B	9	57	B	9	51	B	13	82
COM	D	34	440	D	10	139	B	14	101	B	14	92	B	9	15
SAT	B	28	256	B	24	145	O	13	304	O	13	309	O	18	1269

Table 1
Summary of results for CKT1 and CKT2

5 Interpretation of the Experimental Results

5.1 Stage 1: Summary of Results with respect to Bug Finding

Tables 1-2 compare the bug-finding effectiveness of all the subsetting heuristics with different approximation thresholds. In the case of CKT1 and CKT3, no results are reported for the threshold of 200000 BDD nodes, since the intermediate frontiers until hitting the bug do not grow that large. The letters in the column R stand for the three possible outcomes of the verification:

- B - the bug is found.
- D - the verification is blocked in a dead-end (i.e., a *false* fixed-point is reached).
- O - a time-out, or memory-out has occurred.

The other two columns record the iteration in which the bug/dead-end is reached or the state explosion occurs, and the corresponding CPU time in seconds. The numbers for memory consumption are omitted from these tables, as they follow the same pattern as the CPU time.

Exhaustive vs. semi-exhaustive. As shown in the EXH line the bug can also be found by fully exhaustive verification in 3 out of the 4 circuits. This is due to our selection of verification problems where a bug was known to exist. Only in CKT4 (in which the bug was artificially introduced) exhaustive verification runs out of memory.

Nevertheless, most of the approximation heuristics prove themselves by consistently finding the bug too, even in the cases (CKT4) where exhaustive verification fails to do so. More importantly, when successful, semi-exhaustive algorithms significantly outperform the exhaustive one in terms of CPU time.

Ckt	CKT3, 141 latches, 34 inputs						CKT4, 72 latches, 26 inputs								
thresh	100000 nodes			50000 nodes			200000 nodes			100000 nodes			50000 nodes		
	R	iter	time	R	iter	time	R	iter	time	R	iter	time	R	iter	time
EXH	B	24	120	B	24	120	O	12	640						
SP	B	24	121	D	41	158	B	25	671	B	29	409	B	33	227
HB	B	24	115	B	24	113	D	14	234	D	10	79	D	10	71
UA+	B	24	118	B	24	129	O	12	522	O	12	341	O	12	375
RUA+	B	24	118	B	24	123	O	12	478	O	12	335	O	12	365
UA	B	24	117	B	30	119	B	23	876	B	23	532	B	23	395
RUA	B	24	118	B	30	120	D	47	981	D	70	732	D	55	496
UA-	B	24	121	B	24	118	D	14	465	D	10	94	D	10	78
RUA-	B	24	121	D	31	138	D	22	415	D	58	567	D	22	123
COM	D	26	120	D	24	107	B	23	384	B	23	198	D	170	996
SAT	B	34	156	D	34	133	D	10	124	D	8	60	D	6	39

Table 2
Summary of results for CKT3 and CKT4

Impact of the threshold. Increasing the aggressiveness of the approximation (by lowering the threshold to 100000 or 50000 nodes) rarely prevents the finding of the bug. At most, the lower threshold causes the verification to miss the closest bugs and perform a few more iterations until hitting the next ones. Quite surprisingly, even then, the overall time stays much smaller due to the speed of performing single steps on smaller frontiers.

A quite unusual effect can be observed in the case of the UA/RUA heuristics for CKT2. Lowering the threshold from 100000 to 50000 nodes helps in finding the bug in less iterations. This shows that the first, more conservative, approximation is not necessarily the one that preserves the interesting, buggy states.

Impact of the quality. The quality factor is responsible for the fine tuning of the UA/RUA approximations. Lowering the quality (from UA+ to UA and then to UA-) leads to more aggressive approximations, even to the extent of degrading the density of the result (in the case of UA-). However, here more aggressive does not necessarily mean better. Indeed, UA- is definitely worse, leading to dead ends in many cases, and there is no clear winner between UA+ and UA.

Comparing heuristics. The SP and UA heuristics seem to be the overall winners, both of them finding the bug in 9 out of the10 cases. As we will justify later, this is due to their high conservativeness. At the opposite side of the spectrum we find the highly aggressive heuristics COM and SAT that rarely find the bug, but when they do they are much faster than SP or UA.

In the absence of additional data, these observations are necessarily speculative. The next two sections will allow us to make more educated comments based on the statistics collected along the run and the grading of the approximated frontiers.

Ckt	CKT3, threshold 50000 nodes							CKT4, threshold 50000 nodes						
	R	iter	time	N	T	St	Nd	R	iter	time	N	T	St	Nd
EXH	B	24	120	-	-	-	-	O	12	640	-	-	-	-
SP	D	41	158	4	0.7	92	78	B	33	227	14	1.4	52	41
HB	B	24	113	3	0.5	59	73	D	10	71	3	0.9	20	48
UA+	B	24	129	3	4.8	88	79	O	12	375	5	17	92	87
RUA+	B	24	123	3	3.4	98	85	O	12	365	5	17	92	87
UA	B	30	119	2	1.4	65	51	B	23	395	9	13	64	29
RUA	B	30	120	2	1.4	82	51	D	55	496	23	6.7	68	43
UA-	B	24	118	3	2.0	59	73	D	10	78	3	5.8	16	49
RUA-	D	31	138	2	1.4	75	79	D	22	123	6	5.4	22	57
COM	D	24	107	1	2.1	52	4	D	170	996	48	3.0	43	16
SAT	D	34	133	2	1.6	e-13	26	D	6	39	1	.01	e-8	0.8

Table 3
Statistics collected for circuits CKT3 and CKT4 with threshold = 50000 nodes

5.2 Stage2: Statistics Collected Along the Run

Due to space limitations, we present only the statistics collected in two representative cases: the verification of CKT3 and CKT4 with a threshold of 50000 nodes. Table 3 reports the corresponding data, where the additional column *N* stands for the number of approximations, while *T*, *St*, and *Nd* are shortnames for the averages *AvgT* (in seconds), *AvgSt* and *AvgNd* (in percentages) respectively (see section 4.2).

The data for CKT3 is characteristic of medium complexity problems, where the bug or the dead-end is reached relatively fast and thus only a few approximations are required. In contrast, the verification of CKT4 involves more approximations, and its statistics reflect a slightly different picture.

Aggressiveness. In the previous section we discussed aggressiveness as a function of the threshold or the quality, and its impact on bug finding. In a different dimension, we see here the aggressiveness as a function of the heuristic, and we effectively measure it through the average quantities *St* and *Nd*.

UA+ and RUA+ seem to be the most conservative (i.e. least aggressive) approximations - keeping at least 90% of the states - due to the high value of the quality factor. Sometimes even too conservative (see the data for CKT4), leading to the same blow-up in memory encountered in the fully exhaustive verification.

SP is also quite conservative when only a few approximations are involved (CKT3), but on the long run (CKT4) SP gets more relaxed losing as much as 50% of the states. In contrast, both UA and RUA are more predictable, consistently keeping 60-70% of the states, and thus performing better on the long run.

HB, UA- and RUA- are much less careful, often throwing more than 50%

of the states and occasionally losing in density - this comes as a surprise in the case of HB whose explicit purpose is to increase the density of the result.

This is not the case for COM, which definitely achieves the best density, throwing away much more nodes than states (usually one order more). However, often COM throws away too many states, ending up in dead ends. We attribute its aggressiveness to the fact that it applies RUA (on the result of SP) with a threshold of 0 nodes, thus allowing an arbitrarily large reduction.

Even more aggressive is SAT, which keeps only a few hundreds of states and nodes. This is often bound to fail, unless the saturated variables are chosen very carefully or the frontiers are very dense in bugs (as it seems to be the case for CKT1).

Speed. The overall time for finding the bug depends not only on the quality of the approximations and the number of iterations, but also on the time required by the approximations themselves. It is this last factor that we measure in the column *T*.

The numbers here confirm the theoretical predictions. HB and SP (in this order) are the fastest heuristics, their time complexity being linear in the size of the BDD. On the other hand both UA and RUA have a quadratic worst-case complexity [5], and thus are often significantly slower. It is this factor that accounts for the difference in CKT4 where SP finishes much faster than UA, in spite of performing 10 more iterations. Also, achieving higher quality takes more time, as UA+ is slightly slower than UA, which on its turn is slower than UA-.

As a combination of SP and RUA, COM inherits the quadratic complexity of RUA. Finally, SAT is the most expensive approximation (based on the data for all the cases, and not just Table 3). Indeed, its algorithm requires an operation of existential quantification at the level of each variable that is not saturated. As a consequence, there is a high price to pay for the SAT when applied to large BDDs.

5.3 Stage3: Grading the Approximated Frontiers

Out of the four examples, CKT3 was the only one for which backward analysis completes, and for which we were able to compute the target states. Table 4 reports the data obtained by grading the approximated frontiers for various heuristics with a threshold of 50000 nodes. The columns *R*, *iter*, *time* and *N* are as in Table 3. The next columns contain couples of the form (i_k, $Grade_k$) with i_k being the iteration of the k-th approximation,and $Grade_k$ the grade of the corresponding approximated frontier.

Comparing SP with HB, we note that SP starts better conserving more target states in the iterations 21 and 22, but HB wins in the iteration 23 (the last one before the bug) keeping nearly 4X more target states than SP. This accounts for the fact that HB hits the bug in the iteration 24, while SP misses it and eventually enters in a dead-end.

Ckt	CKT3, threshold 50000 nodes											
	R	iter	time	N	i_1,	$Grade_1$	i_2,	$Grade_2$	i_3,	$Grade_3$	i_4,	$Grade_4$
EXH	B	24	120	-								
SP	D	41	158	4	21,	4.7e-34	22,	2.6e-35	23,	1.9e-34	24,	1.3e-35
HB	B	24	113	3	21,	3.5e-34	22,	1.5e-35	23,	6.4e-34		
UA	B	30	119	2	21,	3.5e-34	22,	9.4e-36				
RUA	B	30	120	2	21,	3.5e-34	22,	1.5e-35				
COM	D	24	107	1	21,	2.6e-34						
SAT	D	34	133	2	21,	8.4e-47	29,	1.1e-46				

Table 4
Grades of approximated frontiers for CKT3 with threshold = 50000 nodes

UA gets the same grade with HB in the iteration 21, but then it loses more target states than HB in the iteration 22, and thus misses the closest bug in the iteration 24. Nevertheless, it eventually comes back and hits the next bug in the iteration 30.

Even more interesting is the case of RUA, which apparently runs head to head with HB up to the iteration 23, and still misses the bug in the iteration 24. RUA and HB keep indeed the same number of target states in the iteration 23, but not necessarily the same states. HB manages to keep some of the states at distance 1 from the bug, while RUA throws all of them away. This suggests a refinement of our grading function that would weight the target states according to their distance from the bug, i.e. assigning a higher grade to target states closer to the bug.

Finally, SAT and COM get the lowest grade at the iteration 21, both losing too many target states from the very first approximation. Moreover, the few target states that remain are not close enough to the bug, so advancing a few more iterations does not suffice for reaching any of the buggy states.

6 Conclusions and Future Work

Previous studies [1,3,5,9] advocate the merits of semi-exhaustive techniques on the basis of their coverage of the reachable state space and the density of the approximated frontiers. The main contribution of our work is to evaluate these techniques with respect to bug finding. Our first results confirm the potential of semi-exhaustive verification as a bug hunting tool.

Beyond the point of view of a casual user, we also try to gain more insight into the behavior of the semi-exhaustive algorithms. For this purpose, we collect statistic data on the verification runs and analyze it along several dimensions. This analysis provides a better understanding of the various subsetting heuristics, and in particular of the tradeoffs to be considered for each heuristic.

A shortcoming of most the subsetting heuristics is that they are hard to tune for bug finding, since they are not guided in any way by the specification

that is checked. The only exception in this sense is the SAT heuristic, but our results with SAT were quite disappointing. It might be that our choice of using SAT with the specification variables is not the best one, but we rather tend to blame the algorithm of SAT which makes too strong assumptions on the design that is verified. We believe that refining the SAT algorithm in this direction, as also suggested in [5], would be beneficial with respect to bug hunting.

Acknowledgements. We would like to thank Amitai Irron for his great work on architecting and developing the formal verification platform in which we made the evaluation. Boris Ginsburg's work on the evaluation of dense frontier subsetting with respect to covered states has inspired this work.

References

[1] K.Ravi, F. Somenzi. High Density Reachability Analysis. In Proceedings of International Conference on Computer-Aided Design, Santa Clara, CA, November 1995.

[2] M.Ganai, A Aziz. Efficient Coverage Directed State Space Search. In Proceedings of IWLS'98.

[3] J. Yuan, J.Shen, J.Abraham, and A.Aziz. On Combining Formal and Informal Verification. In Proceedings of the Computer Aided Verification Conf., July 1997.

[4] C.Yang, D.Dill. Validation with Guided Search of the State Space. In Proceedings of DAC'98.

[5] A.Aziz, J.Kukula, T. Shiple. Hybrid Verification Using Saturated Simulation. In Proceedings of DAC'98.

[6] R.Bryant. Graph-based Algorithms for Boolean Function Manipulations. IEEE Transactions on Computers,C-35:677-691, August 1986.

[7] K.L. McMillan. Symbolic Model Checking. Kluwer 1993.

[8] T.R. Shiple. Formal Analysis of Synchronous Circuits PhD thesis, University of California at Berkeley, 1996.

[9] K. Ravi, K.L. McMillan, T.R. Shiple, F. Somenzi. Approximation and Decomposition of Binary Decision Diagrams. In Proceedings of DAC'98.

[10] O. Coudert, J. Madre. A Unified Framework for the Formal Verification of Sequential Circuits. In Proceedings of ICCAD'90.

[11] B. Lin, R. Newton. Implicit Manipulation of Equivalence Classes Using Binary Decision Diagrams. In Proceedings of ICCD'91.