Improving the creation of AIGs from reactive synthesis

FLORIS HEINEN, University of Twente, The Netherlands

In the SYNTCOMP competition, correct hardware independent logic networks in the form of AIGs need to be created from logic descriptions. Knor, a participating program that creates such solutions, converts during a final stage its BDD to AIG which can be optimized. In this paper multiple AIG minimization techniques are explored and benchmarked. A generally applicable AIG minimization strategy is found with the benchmarks, and the possibility of merging this with the initial creation of the AIG is discussed.

Additional Key Words and Phrases: Reactive synthesis, And-Inverter Graphs, Knor, ABC

1 INTRODUCTION

SYNTCOMP is a competition in the process of reactive synthesis, with the aim to foster research in new tools for automatic synthesis of systems [4]. Reactive synthesis, or the creation of state machines from specifications such as Linear Temporal Logic (LTL) formulas, can be solved with the help of parity games in one of SYNTCOMP's categories. Here, the participating reactive synthesis tools must present their solutions as an And-Inverter Graph (AIG). The tools are then ranked by the number of given problems they can solve within a time limit. An additional ranking exists for quality of solutions, based on the size of the solutions by counting the amount of logic AND gates and latches.

Two participants, Knor[10] and Strix[5], compete in this parity game synthesis category. In order to gain points in the secondary 'quality' ranking, both try to minimize their resulting AIG with the ABC tool[1], at the cost of longer calculation time. This ABC tool hosts a plethora of commands to modify AIGs, ranging from balancing to complete AIG rewriting. The two participants chain these commands in the hope of restructuring the AIG in such a way that the network shrinks. Due to the lack of research on minimizing AIGs created from parity games and the shear amount of different commands ABC hosts, finding a single generic minimization strategy for parity game solutions is tricky. Additionally, the optimization strategies ABC offers are not focused on solutions of parity games.

This research therefore has two purposes. Firstly, to give better insight in the overall process of reactive synthesis. Secondly, to research and explore different ABC commands to create a general optimization strategy, including their strengths and weaknesses. This way, future SYNTCOMP participants can expand upon the insight we hope to create and improve upon this new optimization strategy.

We will try to answer the following three questions. One, which ABC commands work best in which situations? Two, how can these commands be chained for a generally optimal minimization strategy? And three, can these minimization strategies be combined with the creation of AIGs instead of applying them afterwards?

The structure of this paper is as follows: After this introduction the prerequisite knowledge is found, where some basic concepts are briefly explained. Then the methodology of the tests to be performed is explained. Afterwards, the results will be extensively explained and discussed. Lastly, in the Future works section, possible improvements are discussed.

2 PRELIMINARIES

We explain the following logic networks due to their relevance to this paper: AIGs and BDDs.

2.1 And-inverter graph

AIGs are a directed a-cyclic graph of logic AND and inverter gates. Due to the possibility of redundancy, different AIGs can have the same solution that function exactly the same as seen in figure 1. This shows that minimization algorithms could reduce the AIG size while maintaining the same functionality.

In Figure 1, two AIGs are shown, both representing the same Boolean function. The circles as nodes represent an Boolean AND, with the lines as edges connecting them. The SYNTCOMP and this research ignore the cost of inverter gates, and focus solely on AND gate and latch count, resulting in the right graph receiving a higher score on AIG size.

2.2 Binary decision diagram

A BDD is a Boolean logic network made out of nodes that represent a single input variable. Each node contains two outgoing edges, each describing the path the network takes if the input variable is true or false. Just like AIGs, BDDs can also be written differently while having the same functionality, as shown in figure 2.

3 RELATED WORK

The SIS tool[2] tries to minimize AIGs globally. Downsides to this approach are the number of hand-tuning and trial and error necessary to gain a beneficial result. The ABC tool[7] uses a local transformations to minimize an AIG. Even though this means that the result is still sub-optimal after a single iteration, multiple iterations are possible due to the significant decrease in calculation time. Multiple iterations practically result in a global minimization, leading to a technique that achieves smaller AIGs in less time than the former global approach. Another AIG optimization technique is balancing. Although this might not help us in the rankings of SYNTCOMP, they do increase the amount of branches of an AIG and reduce the depth[6]. Starting out with such an AIG might improve the amount of iterations locally altering algorithms need. The process of converting a BDD into an AIG has been studied in [9]. This paper explores converting to an AIG in Irredundant Sum Of Products(ISOP) format in the hope of creating a small AIG. Even though this did not result

TScIT 40, February 2, 2024, Enschede, The Netherlands

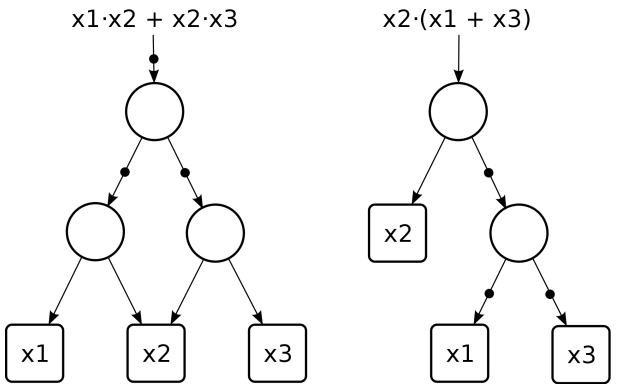

Fig. 1. Two And-Inverter graphs, both representing the same Boolean function. Circle nodes represent AND gates, and edges with a dot are inverted signals. The leftmost AND node can be removed by inverting the edges of the rightmost AND node and its own output edge.

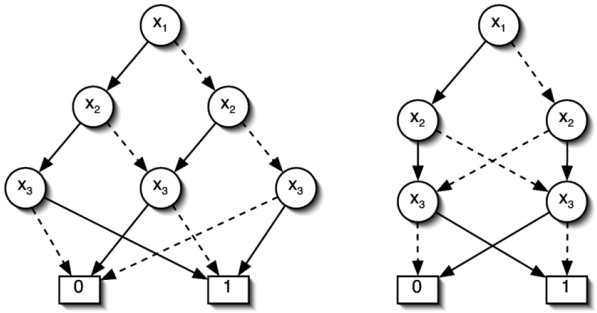

Fig. 2. Two BDDs, both representing the same Boolean function, but the right requiring one less node. When a circular node's value holds, the solid edge is followed. The dashed line is followed otherwise.

in generally smaller AIGs, it does result in a more balanced AIG. Minimizing after this technique has been applied, might decrease calculation time when compared to separately generating an AIG, balancing it and then minimizing it.
BDD minimization will reduce the necessary time to convert, as no time will be spent on redundant branches. These minimization techniques have existed for a while[8]. This technique does not consider properties these BDDs might have if they are generated from parity games.

4 METHODOLOGY

If we want to figure out what the best way of getting the smallest AIG throughout the whole synthesis process, we will need to go through each possibility in that process as well. In this section, we explain the exact process of the testing and its scope. We start by briefly explaining the testing setup and corresponding flow. Then, we go into further detail of each part of the process. Finally, we explain the hardware on which the tests are run.

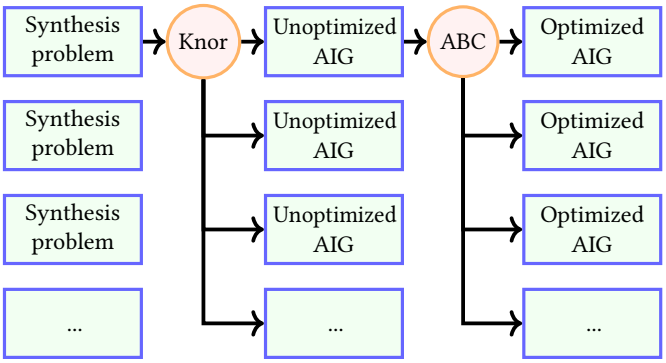

Fig. 3. Synthesis flow. Each synthesis problem can be solved with Knor into an unoptimized AIG in multiple ways. Each unoptimized AIG can in turn be optimized with ABC in multiple ways.

4.1 The test setup and flow

At the origin lies a synthesis problem. The Knor program solves these synthesis problems and outputs an AIG. The output AIG will be further optimized with the help of ABC. The process is visualized in figure 3. The figure shows how Knor solves a single synthesis problem into multiple different unoptimized AIGs. Then, for each unoptimized AIG, we will use ABC to test different optimization strategies, resulting in different optimized AIGs.

4.2 Synthesis problem selection

Due to time limitations, we had to limit out scope to a subset of available synthesis problems picked from the SYNTCOMP benchmark repository [1].

The selection consists of 22 randomly selected synthesis problems, with the requirement that they are realizable. Otherwise, we ofcourse could not have tested and optimized the result. The selected problems are listed in table 1

4.3 Solving with Knor

Knor is part of our first test as it creates the unoptimized AIG files. Knor's program flags modify the way in which it creates the AIGs. Because these influence the structure of the resulting AIGs, they might also impact the results of ABC optimizations. Therefore, the different Knor arguments need to be tested in combination with ABC optimizations to draw conclusions. Two types of Knor flags can be chosen: solve flags and synthesis flags.

Solve flags indicate which solve algorithm is used. A large number of algorithms is available [2] so to limit the scope of this research, we limit our solve flags to the recommended algorithms. The solve flags picked for testing are described in table 5. Only a single solve flag can be used per Knor command, leading us to test for 5 possible solve flags.

Synthesis flags are used for picking the way in which Knor's internal strategy BDD gets converted into an AIG. The three flags we will use for testing are '–no-bisim', '–binary' and '–isop', as described in table 4. The '–compress' argument will perform basic AIG

[1] https://github.com/SYNTCOMP/benchmarks
[2] https://github.com/trolando/oink

Table 1. List of all randomly selected synthesis problems used in our testing.

Filename
ActionConverter.tlsf.ehoa
EscalatorCountingInit.tlsf.ehoa
MusicAppFeedback.tlsf.ehoa
Radarboard.tlsf.ehoa
SPIWriteSdi.tlsf.ehoa
SliderScored.tlsf.ehoa
TwoCounters4.tlsf.ehoa
TwoCountersInRangeA6.tlsf.ehoa
amba_decomposed_arbiter_5.tlsf.ehoa
amba_decomposed_encode_13.tlsf.ehoa
amba_decomposed_encode_8.tlsf.ehoa
amba_decomposed_lock_3.tlsf.ehoa
amba_decomposed_tsingle.tlsf.ehoa
full_arbiter_5.tlsf.ehoa
lilydemo21.tlsf.ehoa
loadcomp5.tlsf.ehoa
ltl2dba06.tlsf.ehoa
ltl2dba16.tlsf.ehoa
ltl2dba26.tlsf.ehoa
ltl2dpa01.tlsf.ehoa
ltl2dpa11.tlsf.ehoa
ltl2dpa21.tlsf.ehoa

compression after solving, so to not influence the ABC minimization commands, we exclude this flag. The '–best' flag will choose for the user which of the three initially synthesis discussed flags should be used. This flag is also excluded, as one strategy might be beneficial for the short term, but disadvantageous on the long term, so we need to test for all. These picked synthesis flags can be combined for a different result, resulting in $2^3 = 8$ possible synthesis flag combinations.

With 5 possible solve flags, combined with 8 possible synthesis flag combinations, we take $5 * 8 = 40$ different Knor commands into account for our tests. This results in 40 different unoptimized AIGs per selected synthesis problem.

4.4 Optimizing with ABC

We use ABC to optimize our AIGs that result from using Knor. ABC offers many commands, and the ones that seem promising for our purpose of decreasing the amount of AND gates in AIGs are summed up in table 2. We selected the promising commands based on their name, description and code documentation, with the requirement that they should be aimed towards reducing the amount of AND nodes. In addition, commands that might positively modify AIGs without reducing the amount of AND nodes, but create new optimize opportunities for the other commands are shown table 3. This includes graph balancing commands and redundancy removal commands. Both tables contains ABC command arguments with their description and corresponding command flags. Command flags like '-h' for printing help information and '-v' for verbose messaging are left out, as these do not influence the results. Number signs ('#') indicate that the flag requires a number to be used with. To find optimal chaining strategies, we cannot go over every ABC optimization combination, as the amount of possibilities grows exponentially, while some predefined strategies contain over 15 commands. Therefore, for Test 1, we start out by comparing the effectiveness of each individual ABC command. For Test 2, we test if the cleanup arguments have effect on the unoptimized AIGs. Then, for Test 3, we will devise some basic strategies and compare them with the predefined ones.

4.5 Testing in python

A python script has been devised that can perform the previously mentioned tests. It is able to generate a list of all possible flags for both Knor and ABC. These flag combinations are then used to create both solutions to the synthesis problems as well as optimizations on the solutions. The AIG solutions of all solved synthesis problems are saved. Their paths and statistics like their amount of AND gates and the time necessary to solve the synthesis problem are stored inside a JSON[3] file. This way, when we want to optimize a specific solution, we can find its path back by searching in the JSON file for the matching solution. The outputs of AIG optimizations are saved, with their corresponding statistics in the JSON file as well. When a chain of optimizations is issued to the script, it will save each intermediate result as if it were an individual optimization command. This way, these intermediate results can be reused for other optimizations that share their first part, like 'compress' and 'compress2'.

4.6 Performed tests

In the first test we compare each ABC optimization. The script performs every argument including many variations of them on each synthesis solution. The results are discussed in 5.2. The second test is similar to the first test, but aimed towards ABC cleanup related commands according to their description. The third test compares all predefined optimization strategies.

5 RESULTS

In this section we discuss the results of each test. Plots are shown as letter-value plots, where each successive block outwards represents only half of the remaining data. This plot type is picked as this variation of a boxplot show more information about the distribution and works well on smaller sample sizes [3].

5.1 Knor solve results

All synthesis problem files have been solved with each possible Knor combination within the scope of this paper. First, we collected all smallest AIGs per synthesis problem. Then, per Knor argument combination, we compare the outcome to the minimum of that file, where a gain of 1 means that the resulting AIG is equally big as the smallest AIG of this we could create. A gain of 2 means the result is twice as big by having twice as twice as many AND gates. All gains are averaged per Knor argument over each synthesis problem. The results are shown in figure 8. For visibility, the top part of the graph is cut off, as our focus is on Knor flag combinations that are as close to 1 as possible. The graph shows that using no synthesize flags or

[3] https://www.json.org/json-en.html

Table 2. Tested ABC optimize commands with their flags

Command	Description	Flags
balance	transforms the current network into a well-balanced AIG	[-ldsx]
rewrite	performs technology-independent rewriting of the AIG	[-lz]
drw	performs combinational AIG rewriting	[-C #] [-N #] [-lzr]
refactor	performs technology-independent refactoring of the AIG	[-N #] [-lz]
drf	performs combinational AIG refactoring	[-M #] [-K #] [-C #] [-elz]
drwsat	performs combinational AIG optimization for SAT	[-b]
resub	performs technology-independent restructuring of the AIG	[-KNF #] [-lz]
dc2	performs combinational AIG optimization	[-blfp]
irw	perform combinational AIG rewriting	[-lz]
irws	perform sequential AIG rewriting	[-z]
iresyn	performs combinational resynthesis	[-l]

Table 3. Tested ABC cleanup commands with their flags

Command	Description	Flags
b	Transforms the current network into a well-balanced AIG	[-ldsx]
trim	Removes POs def by constants and PIs wo fanouts	[]
cleanup	Removes PIs w/o fanout and POs driven by const-0	[-io]
scleanup	Performs sequential cleanup of the current network	[-cenm] [-F #] [-S #]
csweep	Performs cut sweeping using a new method	[-C #] [-K #]
ssweep	performs sequential sweep using K-step induction	[]
scorr	Performs sequential sweep using K-step induction	[]

Table 4. Knor synthesis arguments

Parameter	Description
–bisim	Minimize the state space using bisimulation minimization prior to synthesis
–onehot	Encode the states using one-hot encoding instead of logarithmic encoding
–isop	Use ZDD covers for the conversion to AIG

Table 5. Knor solving arguments

Parameter	Description
–sym	Internal symbolic parity game solver
–tl	Tangle learning
–rtl	Recursive tangle learning
–fpi	Distraction Fixpoint Iteration
–zlk	Zielonka's recursive algorithm

only the '–isop' results in a consistently small AIG, irrelevant of Knor solve flags. Additionally, it shows that combining the '–no-bisim' and '–binary' arguments result in a significantly larger AIG. Only using these Knor flags throughout the rest of the benchmarking would be unwise, as we cannot exclude the possibility yet that a larger initial AIG might lead to a better optimization later with ABC commands.

5.2 Test 1

The number of AND gates of each ABC optimization command result is compared to the number of AND gates the solution has by calculating the decrease in size:

$$Gain = \frac{solution\ AND\ count}{optimized\ AND\ count}$$

Comparing optimizations through their gain instead of amount of AND gates reduced allows us to compare the effectiveness of each command regardless of the initial AIG size. Otherwise, using a relatively little effective approach that reduces 1000 AND nodes to 950 would seem equally effective as a relatively more effective reduction from 100 to 50 AND nodes, as both remove 50. The results are plotted in figure 4. In the letter-value plot, only the best 15 optimizations are shown for brevity, and are sorted from highest median to lowest median, as can be seen from the gradual decline in

Table 6. ABC optimization strategies

Optimization strategy	Arguments
c2rs	b -l, rs -K 6 -l, rw -l, rs -K 6 -N 2 -l, rf -l, rs -K 8 -l, b -l, rs -K 8 -N 2 -l, rw -l, rs -K 10 -l, rwz -l, rs -K 10 -N 2 -l, b -l, rs -K 12 -l, rfz -l, rs -K 12 -N 2 -l, rwz -l, b -l
compress	b -l, rw -l, rwz -l, b -l, rwz -l, b -l
compress2	b -l, rw -l, rf -l, b -l, rw -l, rwz -l, b -l, rfz -l, rwz -l, b -l
compress2rs	b -l, rs -K 6 -l, rw -l, rs -K 6 -N 2 -l, rf -l, rs -K 8 -l, b -l, rs -K 8 -N 2 -l, rw -l, rs -K 10 -l, rwz -l, rs -K 10 -N 2 -l, b -l, rs -K 12 -l, rfz -l, rs -K 12 -N 2 -l, rwz -l, b -l
drwsat2	st, drw, b -l, drw, drf, ifraig -C 20, drw, b -l, drw, drf
r2rs	b, rs -K 6, rw, rs -K 6 -N 2, rf, rs -K 8, b, rs -K 8 -N 2, rw, rs -K 10, rwz, rs -K 10 -N 2, b, rs -K 12, rfz, rs -K 12 -N 2, rwz, b
resyn	b, rw, rwz, b, rwz, b
resyn2	b, rw, rf, b, rw, rwz, b, rfz, rwz, b
resyn2a	b, rw, b, rw, rwz, b, rwz, b
resyn2rs	b, rs -K 6, rw, rs -K 6 -N 2, rf, rs -K 8, b, rs -K 8 -N 2, rw, rs -K 10, rwz, rs -K 10 -N 2, b, rs -K 12, rfz, rs -K 12 -N 2, rwz, b
resyn3	b, rs, rs -K 6, b, rsz, rsz -K 6, b, rsz -K 5, b
rwsat	st, rw -l, b -l, rw -l, rf -l
src_rs	st, rs -K 6 -N 2 -l, rs -K 9 -N 2 -l, rs -K 12 -N 2 -l
src_rw	st, rw -l, rwz -l, rwz -l
src_rws	st, rw -l, rs -K 6 -N 2 -l, rwz -l, rs -K 9 -N 2 -l, rwz -l, rs -K 12 -N 2 -l

median line per letter-value bar. Median was picked over mean, as the lower extreme values all approached 1, whereas higher extreme values were boundless. This means grouping and sorting by the mean would give a skewed result. The ABC optimization argument 'dc2' shows most promise, with a median gain around 1.4, meaning it minimizes unoptimized AIGs around 28%. The '-b' and '-l' flags for the 'dc2' optimization show no significant direct effect, but the '-f' flag seems to negatively influence the performance of this optimization. As for the 'rs' command, the unifying factor seems to be the '-K' flag, which is the highest allowed number by ABC. It makes sense that this results in the better performance compared to the other 'rs' commands, as the 'K' flag dictates the maximum cut size it can use, which influences the amount of ways it can find new and improved AIG structures. A similar thing can be said for the '-N' flag, where a higher number seems to allow the 'rs' command to look for more opportunities.

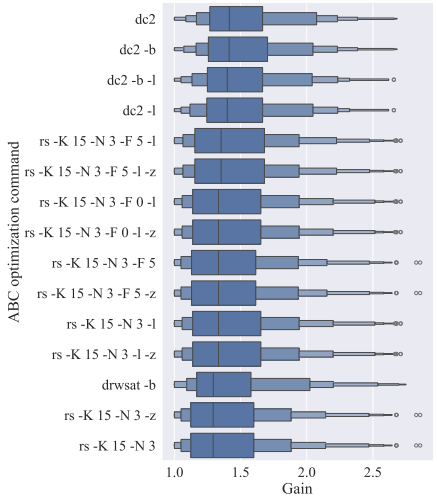

Fig. 4. Effectiveness of the top 15 ABC optimization commands, expressed in gain compared to the solution that is optimized. Most effective is 'dc2' without the '-f' flag, followed by the 'rs' commands with a '-K 15' and '-N 3' flag.

5.3 Test 2

Similar to Test 1, we compare the resulting amount of AND nodes of each cleaned-up AIG to the amount of AND nodes available in the synthesis solution. In contrast to Test 1, instead of using the median to sort the arguments on effectiveness, we went for sorting based on the mean. This is because only 'scorr' and 'ssweep' showed a median gain higher than 1. They still appear on top if we sort on mean, but now we can compare the other cleanup commands as well. It is interesting to see how the cleanup arguments overall decrease the amount of AND nodes less than the optimization commands, but have significantly higher outliers. Lastly, the other commands have a nearly identical effect on the decrease in size for the AIG solutions.

5.4 Test 3

The gains of each optimization strategy have been plotted in Figure 7. For readability purposes, the 15 lines have been separated into two plots. For every step in each strategy, the total gain compared to the original synthesis solution AIG is plotted. This is repeated for every problem file, resulting in a line with corresponding confidence

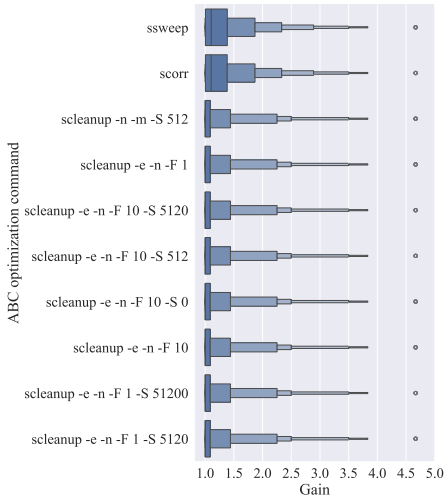

Fig. 5. Effectiveness of the top 10 ABC cleanup commands, expressed in gain compared to the solution that is optimized. Commands 'scorr' and 'ssweep' outperform all others, but still generally pale in comparison to the decrease in amount of AND gates some ABC optimization commands offer.

interval. The ideal strategy would be a steep line that keeps increasing the gain over the original AIG file for every step. Most 'resyn' strategies flatten out quickly, indicating that they lose effectiveness fast. The strategies 'comrpess2rs' and 'resyn2rs' look like they will be able to continue creating higher gains after repeating the strategy. The strategy with the most promise is 'src_rws', as this has almost already created the same amount of gain as the best few others but in only 7 steps.

What is noticeable is that some strategies start with the 'st' command, but according to the ABC documentation, this command converts the given network into an AIG. This means removing this first command already improves the performance of these commands for Knor.

5.5 Test 4

From the previous three tests, we came propose the following optimization strategy: "ssweep, balance, dc2, dc2, balance, dc2, dc2". We chose to start with 'ssweep' to try to remove significant redundancies. It is followed up by repetitions of the best performing 'dc2' optimization and the 'balance' command for structural balancing. The performance of this strategy has been plotted in figure **??**. The new strategy under the name 'custom_0' shows an almost doubling in gain per step. We can conclude this new strategy is therefore a significant improvement in optimization strategy if we are limited to running each strategy once.

6 CONCLUSION AND FUTURE WORKS

Solving synthesis problems comes with many choices to be made. Due to this, trying every possibility to come up with the single best possibility is infeasible. In this paper, we have tested and compared every single promising ABC command, we have compared the already existing optimization strategies, and based on our findings, a new strategy was created that outperforms the other ones by twofold. Further testing will need to be done to truly understand if this new strategy will hold in every situation or if this only holds in the specific circumstances of this study. Overall, the results in this paper can be very useful, giving and improving insight in the general synthesis problem. For future work, the new strategy could be tested if the new gain per step stagnates fast or will slowly decrease. Additionally, because calculation time is also important in the SYNTCOMP, the calculation time of each strategy could be measured and compared as well to find an optimal combination between speed and quality of solutions.

7 REFERENCES

[1] Robert K. Brayton and Alan Mishchenko. 2010. ABC: An Academic Industrial-Strength Verification Tool. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6174)*, Tayssir Touili, Byron Cook, and Paul B. Jackson (Eds.). Springer, 24–40. https://doi.org/10.1007/978-3-642-14295-6_5

[2] SENTOVICH M. E. 1992. SIS: A system for sequential circuit synthesis. *Memorandom no. UCB/ERL M92/41* (1992). https://cir.nii.ac.jp/crid/1574231873788055424

[3] Hadley Wickham Heike Hofmann and Karen Kafadar. 2017. Letter-Value Plots: Boxplots for Large Data. *Journal of Computational and Graphical Statistics* 26, 3 (2017), 469–477. https://doi.org/10.1080/10618600.2017.1305277 arXiv:https://doi.org/10.1080/10618600.2017.1305277

[4] Swen Jacobs, Roderick Bloem, Romain Brenguier, Rüdiger Ehlers, Timotheus Hell, Robert Könighofer, Guillermo A. Pérez, Jean-François Raskin, Leonid Ryzhyk, Ocan Sankur, Martina Seidl, Leander Tentrup, and Adam Walker. 2017. The first reactive synthesis competition (SYNTCOMP 2014). *Int. J. Softw. Tools Technol. Transf.* 19, 3 (2017), 367–390. https://doi.org/10.1007/S10009-016-0416-3

[5] Philipp J. Meyer, Salomon Sickert, and Michael Luttenberger. 2018. Strix: Explicit Reactive Synthesis Strikes Back!. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10981)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 578–586. https://doi.org/10.1007/978-3-319-96145-3_31

[6] Alan Mishchenko, Robert K. Brayton, Stephen Jang, and Victor N. Kravets. 2011. Delay optimization using SOP balancing. In *2011 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2011, San Jose, California, USA, November 7-10, 2011*, Joel R. Phillips, Alan J. Hu, and Helmut Graeb (Eds.). IEEE Computer Society, 375–382. https://doi.org/10.1109/ICCAD.2011.6105357

[7] Alan Mishchenko, Satrajit Chatterjee, and Robert K. Brayton. 2006. DAG-aware AIG rewriting a fresh look at combinational logic synthesis. In *Proceedings of the 43rd Design Automation Conference, DAC 2006, San Francisco, CA, USA, July 24-28, 2006*, Ellen Sentovich (Ed.). ACM, 532–535. https://doi.org/10.1145/1146909.1147048

[8] Arlindo L. Oliveira, Luca P. Carloni, Tiziano Villa, and Alberto L. Sangiovanni-Vincentelli. 1998. Exact Minimization of Binary Decision Diagrams Using Implicit Techniques. *IEEE Trans. Computers* 47, 11 (1998), 1282–1296. https://doi.org/10.1109/12.736442

[9] N. Tomov. 2022. Converting binary decision diagrams to and-inverter graphs using prime-irredundant covers. http://essay.utwente.nl/91688/

[10] Tom van Dijk, Feije van Abbema, and Naum. 2024. Knor: reactive synthesis using Oink. In *TACAS 2024*.

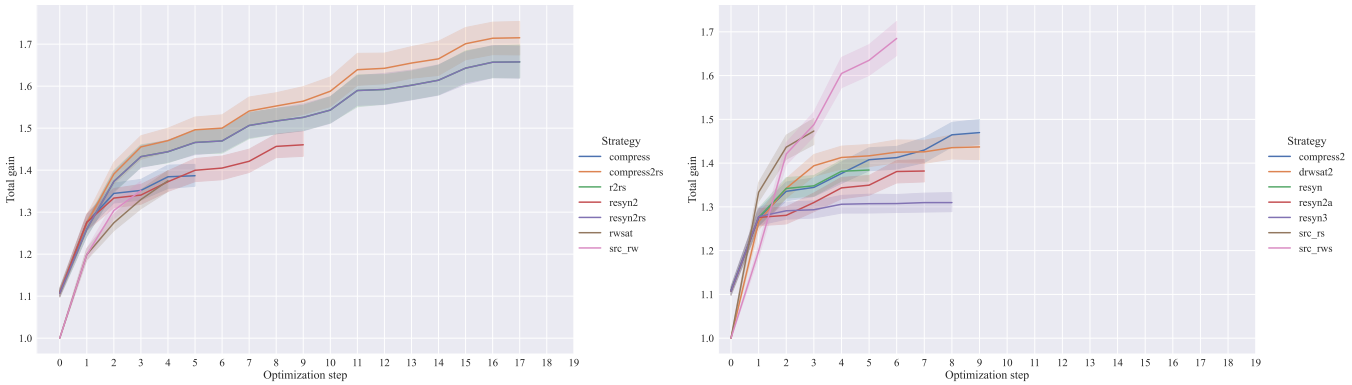

Fig. 6. Gains per step for predefined optimization strategies. Split into two graphs for readability.

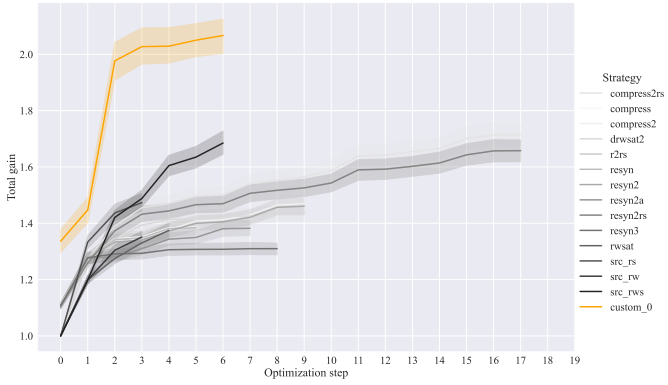

Fig. 7. Compared performance of newly made optimization strategy.

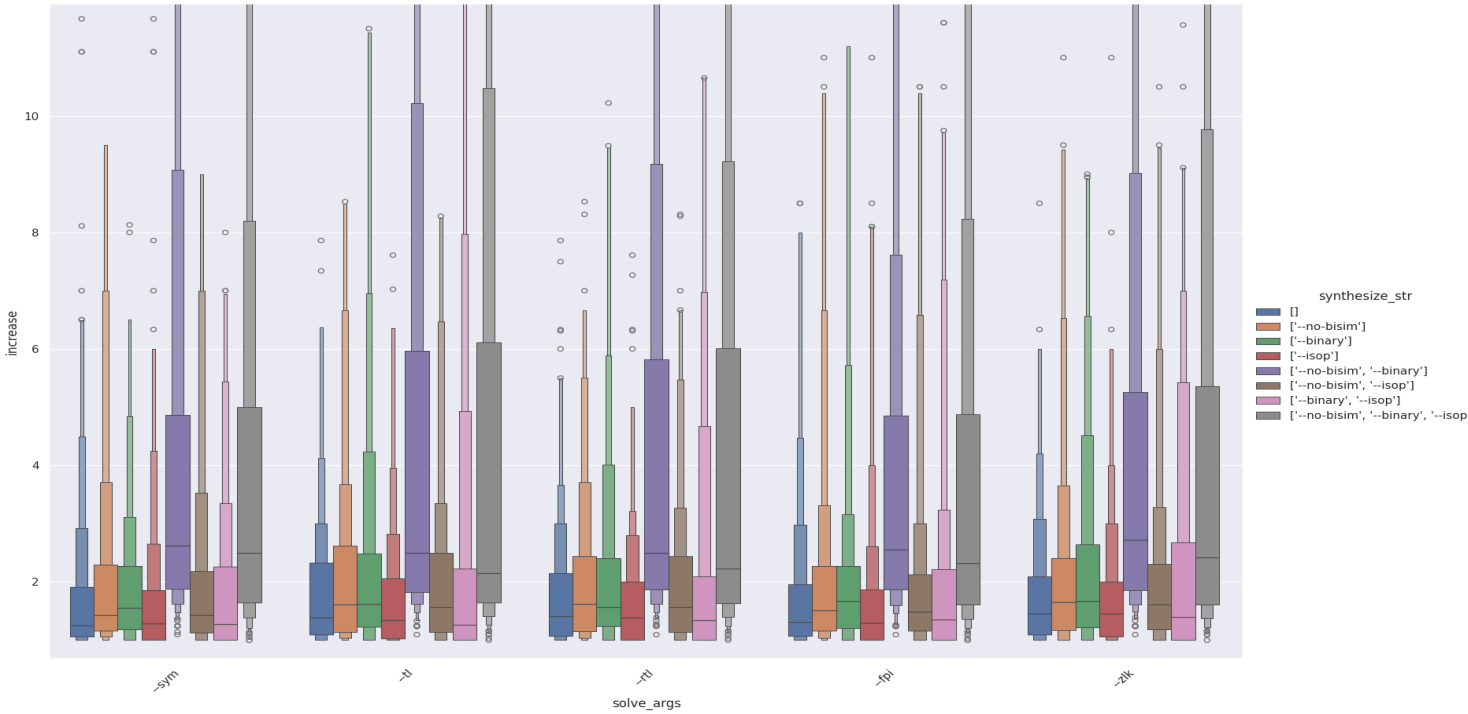

Fig. 8. Zoom-in of letter-value of mean over increase in AIG size. Smaller is better.