Converting Binary Decision Diagrams to And-Inverter Graphs Using Prime-Irredundant Covers

NAUM TOMOV, University of Twente, The Netherlands

Reactive synthesis is the automatic generation of a state machine from a high level specification, such as a linear temporal logic formula. In a recent development in this field, symbolic reasoning using binary decision diagrams has been proposed as a method to drastically reduce computational complexity of reactive synthesis. We have researched a potential improvement to this symbolic reactive synthesis, in which irredundant sum of products is used to convert binary decision diagrams into and-inverter graphs. The principle objective was to reduce the size of the resulting graphs. The results range from sizeable improvements to dramatic setbacks.

Additional Key Words and Phrases: Binary Decision Diagram(BDD), And-Inverter Graph(AIG), Reactive synthesis, Irredundant Sum of Products(ISOP)

1 INTRODUCTION

The field of controller synthesis aims to construct a system that conforms to a given specification. If we can directly synthesize such a system, we do not need to verify its correctness. A novel approach proposed in Remco Abraham's Master Thesis "Symbolic LTL Reactive Synthesis"[1] uses Binary Decision Diagrams in the process of converting a Linear Temporal Logic (LTL) formula into a Mealy machine[6] - a finite-state machine whose output is determined by both its state and its input. This Mealy machine is represented in the AIGER format[2] as this is the format used by the SYNTCOMP competition[4].

In this research, the algorithm we consider is a specific step in this process. This step, at the end of reactive synthesis, is concerned with the generation of the end-product - the and-inverter graph(AIG) that represents the Mealy machine. Currently, a naive approach is employed for this, which leads to sizeable graphs. We experimented with a novel approach to this task, emphasizing the quality of the end-product, measured by its size, and not the performance metrics of the algorithm.

Outline. This paper will first give the reader some context of the field in the Preliminaries section on page 1, then introduce the reader to the specifics of the issue we want to address with this research in the Problem Statement section on page 1. After that, the Methodology is discussed on page 2, followed by an Evaluation on page 4 and Discussion of the results on page 6. At the end of the paper, a Conclusion is provided.

1.1 Problem Statement

In the symbolic reactive synthesis algorithm, the obtained Mealy machine results in a BDD for every output signal and latch of this automaton. It is the translation of this set of BDDs to an AIG that is the topic of interest. Currently, the BDD to AIG conversion happens naively. Unlike BDDs which have canonical forms (for a fixed variable ordering), AIGs that are equivalent in meaning can be vastly different in size. Ideally, we want to generate a graph that is as small as possible. In the ITE conversion, the AIG constructed has up to three AND-gates for each BDD node. A large AIG is sub-optimal because it means a less efficient controller. If we can meaningfully reduce the size of the AIG, we make it more efficient, thereby improving the quality of the controller.

TScIT 37, July 8, 2022, Enschede, The Netherlands

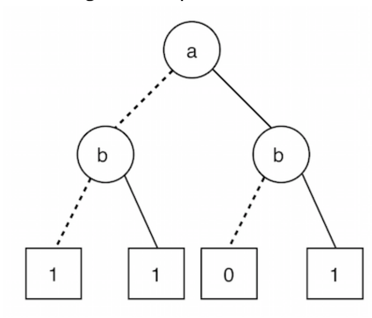
1.2 Contributions

The problem statement leads us to the following research question: Can we reduce the size of resulting graphs when converting Binary Decision Diagrams into And-Inverter Graphs compared to the traditional ITE conversion? This can be broken down into the following sub-questions:

(1) Does using Irredundant Sum of Products generation to convert BDDs to AIGs result in smaller AIGs?
(2) What is the reason for the improvement, or lack thereof?
(3) Can we identify other approaches for reducing the size of the resulting AIG?

2 PRELIMINARIES

Fig. 1. Example BDD $\neg a + b$



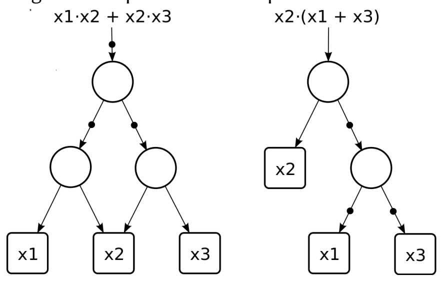
2.1 Binary Decision Diagrams

A binary decision diagram also referred to as BDD is a rooted, acyclic, directed graph used to represent boolean functions[10]. Each node of the graph has 2 outgoing edges, corresponding to true and false. The true edge is a solid line on the right, whereas the false edge is a dotted line on the left. An example BDD can be seen in Figure 1, which represents the boolean expression $\neg a + b$. Each node represents a variable in the formula and the leaves represent $True$ or $False$, which is the output of the formula. BDD generally refers to reduced ordered binary decision diagrams, which are canonical forms of BDDs for a given function and variable ordering. The variable ordering in a diagram plays a major role in its size and finding the optimal ordering (the one that leads to the smallest graph) is known to be an NP-hard problem. For this research, we will not be tackling variable ordering.

2.2 And-Inverter Graph

An and-inverter graph also referred to as AIG, is a directed, acyclic graph that represents a structural implementation of the logical specification of a circuit. As the name suggests, a node can only represent logical conjunction and edges optionally contain markers to indicate negation. Examples of such graphs can be found in Figure 2. The end product of the symbolic LTL synthesis is an AIG, specifically in the AIGER[2] format. This format describes how to represent an AIG with Mealy semantics.

Fig. 2. Example AIGs with inputs on the bottom



2.3 Linear Temporal Logic

Linear temporal logic or LTL, is a branch of boolean logic which has operators that refer to time, hence "temporal", e.g. a condition will *eventually* be true, or a condition will be true *until* another condition is true. An LTL formula represents a specification and is the input used in the process of reactive synthesis.

2.4 Symbolic LTL Reactive Synthesis

Logic synthesis, as previously mentioned, is the generation of a logic gate design from an abstract circuit specification (such as an LTL formula)[1]. The "reactive" in "reactive synthesis" refers to the system being sensitive to changes in the environment, e.g. handling user input. Finally, "symbolic" refers to the fact that binary decision diagrams are used to encode states and manipulate them, without requiring to enumerate each state individually, which has been shown to reduce space and time complexity.

2.5 Irredundant Sum Of Products

An irredundant sum of products is, in boolean logic terms, a disjunction of conjunctions, where none of the products can be altered or removed without changing the formula being represented. They are also known as "prime covers" or "cube sets" and these terms are used interchangeably in this paper. An example of a boolean expression that is prime-irredundant can be found in Table 1, which demonstrates an expression which contains a redundant literal in the first product of the expression, namely y. As shown in the truth table, the latter expression is equivalent, yet more compact. It contains no redundancies, meaning nothing can be removed from it without changing the function. Note that this form is not unique, meaning many ISOP forms exist for a single expression.

An algorithm for devising an ISOP form of a boolean expression is described in the paper "Fast Generation of Prime-Irredundant Covers from Binary Decision Diagrams"[7] by Shin-Ichi Minato. The paper claims that while this form is not minimal, empirically, it is close to the minimal form. On this, we base the assumption that this form will lead to AIGs smaller in size.

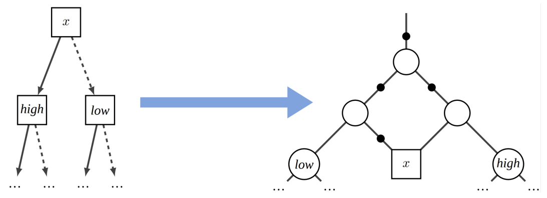
Table 1. Truth table showing equivalence of two expressions, the former of which has a redundant literal 'y' in the first product, and the latter being prime-irredundant

x	y	z	xyz + $(\neg y)x$ Redundant	xz + $(\neg y)x$ Irredundant
0	0	0	*False*	*False*
0	0	1	*False*	*False*
0	1	0	*False*	*False*
0	1	1	*False*	*False*
1	0	0	*True*	*True*
1	0	1	*True*	*True*
1	1	0	*False*	*False*
1	1	1	*True*	*True*

2.6 ITE Conversion

ITE conversion in this paper refers to an If-Then-Else approach to the conversion from BDD to AIG. This is a naive, recursive method where every BDD node is mapped to up to three AIG gates. The resulting AIG could be quite substantial in terms of size, upper bounded by three times the input BDD's amount of nodes. An example can be seen in Figure 3. In the figure, as described by Remco[1], conversion is from BDD (left) to an AIG (right) with high and low nodes labelled to represent the recursive application of this conversion. Each white circle on the right is an AND-gate and each black dot – an inverter. The rectangular node is the input and the edge at the top is the output.

Fig. 3. Image showcasing the ITE conversion from BDD to AIG



3 METHODOLOGY

This research was conducted in three phases - implementation, analysis and documentation. Firstly, the ISOP algorithm needed to be implemented in the repository Sylvan[12], which is a parallel, multi-core binary decision diagram library written in C. Once a working version of this algorithm was derived and thoroughly tested, it had to be integrated into Knor[11]. Knor is a tool for synthesizing

controllers from HOA parity automata. The automata are passed in the form of ".ehoa" files, which stands for extended Hanoi Omega-Automata[9].
A new strategy needed to be developed for using the obtained prime-irredundant cover to generate an AIG. The analysis consisted of studying the results on different examples and trying to identify weak/strong points as well as further improve the conversion. The final phase consisted of logging the results and reasoning as to why they occurred.

3.1 Implementation

For the implementation in Sylvan[12], the ISOP algorithm was implemented as described in Shin-ichi Minato's paper[7]. The pseudocode is shown in Algorithm 1. ISOP is a recursive algorithm, which takes as input a BDD, which is represented by a lower bound L and an upper bound U. The base case for the recursion is when the upper bound is the "true" leaf of a BDD or when the lower bound is the "false" leaf. If neither applies, the variable with the highest order (minvar) in the BDD is extracted and the BDD is decomposed with respect to this minvar. Then, the recursive step takes place and the result is built with the results from the recursive calls to ISOP.
To make use of this algorithm, thorough testing had to be performed, to show that it is correct. For this purpose an inverse algorithm was developed in the same repository. By applying ISOP on some BDD and then reverse ISOP on the result, it can be confirmed that the output of the latter matches the original BDD.

Algorithm 1 Irredundant Sum Of Products

```
Require: $L \implies U$    ▷ lower and upper bounds for the function
  if $L$ is false then return $zdd\_false$
  else if $U$ is true then return $zdd\_true$
  end if
  $v \leftarrow minvar$    ▷ v is input with highest order in BDD
  $Lnv, Unv \leftarrow L(\neg v), U(\neg v)$
      ▷ Lnv and Unv are the false edges of the v node in the BDD
  $Lv, Uv \leftarrow L(v), U(v)$
      ▷ Lv and Uv are the true edges of the node v in the BDD
  $isop0 \leftarrow isop(Lnv, Unv)$    ▷ This is the recursive step
  $isop1 \leftarrow isop(Lv, Uv)$
  $isop* \leftarrow isop(isop0\ \&\ isop1, Uv\ \&\ Unv)$
  $result \leftarrow (\neg v) \cdot isop0 + v \cdot isop1 + isop*$    ▷ Generate cube
  return $result$
```

After the ISOP algorithm was completed and tested, the implementation continued in the Knor[11] repository. A strategy was devised as to how to turn the cover into an AIG, as shown in Algorithm 2, which uses the previously developed ISOP function. Through ISOP, an irredundant prime representation of the BDD is obtained. The advantage of this form is that since it is a sum of products, the translation to AIG is direct. First, you use AND-gates to represent each of the products. Then you invert the outputs of the AND-gates and NAND them to obtain their disjunction. For example, the irredundant sum of products could be the formula $abc + xyz$. We use the *gate_queue*, as seen in Algorithm 2, to use AND-gates to conjugate all the variables in the product. Once the entire product is represented by a single AND-gate, it is added to the *product_gates_queue*, where in the same manner, all products are summed by using three inverters and one AND-gate to obtain a sum of the products, using DeMorgan's law. The final gate obtained from the *product_gate_queue* represents the entire sum of all products. Thus, you have obtained an AIG that conforms to the specification.

Algorithm 2 BDD ISOP cover to AIG

```
  $cover \leftarrow isop(BDD)$
  $products[] \leftarrow enum\_paths(cover)$    ▷ Get all products
  $gate\_queue$    ▷ sub-products stored here
  $product\_gates\_queue$    ▷ finished products stored here
  $i \leftarrow 0$
  while $i \leq len(products)$ do
      $gate\_queue.add(\forall x \in products[i])$
      $i++$
      while $\neg gate\_queue.empty()$ do
          $sub\_prod1 \leftarrow gate\_queue.pop()$
          $sub\_prod2 \leftarrow gate\_queue.pop()$
          $new\_gate \leftarrow create\_AND\_gate(sub\_prod1, sub\_prod2)$
          $gate\_queue.add(new\_gate)$
          if $new\_gate = gate\_queue.back()$ then
                  ▷ last gate represents the entire product
              $last\_gate \leftarrow gate\_queue.pop()$
              $product\_gates\_queue.add(last\_gate)$
          end if
      end while
  end while
  ▷ The product_gates_queue is now initialized with all products
  while $\neg product\_gates\_queue.empty()$ do
      $prod1 \leftarrow product\_gates\_queue.pop()$
      $prod2 \leftarrow product\_gates\_queue.pop()$
      $new\_gate \leftarrow create\_NAND\_gate(\neg prod1, \neg prod2)$
      $product\_gates\_queue.add(new\_gate)$
      if $new\_gate = product\_gates\_queue.back()$ then
              ▷ last gate represents the entire sum
          $res \leftarrow product\_gates\_queue.pop()$
      end if
  end while
  return $res$
```

3.2 Analysis

The Knor[11] repository contains a large number of examples which can be run with a "–sym" option to generate and solve a parity game. The output is in the AIGER format[2] and is generated via a BDD to AIG conversion. The initial implementation of this conversion was a naive, recursive ITE generation, where each BDD node was mapped to up to three gates in the AIG (refer to ITE Conversion for the specifics). To analyze the performance of the new conversion, the old one was used as a benchmark. Note that here "performance" does not refer to the space or time complexity of the algorithm, but rather to the size of the generated graph. We are aiming to reduce

Table 2. Effect of using ISOP conversion on the size of the AIG

Parity Automaton	AND-Gates (ITE)	AND-Gates (ISOP)	AND-Gates (ISOPc)	AND-Gates(ISOPr)
ltl2dba_E	4	4	4	4
ltl2dba_R	5	4	4	4
ltl2dba_alpha	15	20	19	13
amba decomposed lock	16	21	18	18
ltl2dba_Q	17	15	16	13
ltl2dba_U1	19	13	11	12
ltl2dba01	26	24	22	21
lilydemo22	129	155	142	119
full_arbiter	158	162	136	144
amba decomposed tsingle	158	153	140	138
amba decomposed tincr	180	176	168	159
amba_decomposed_encode_10	390	1443	1646	391
SliderDelayed	605	489	489	571
loadfull5	755	772	771	670
amba_decomposed_arbiter	2072	2904	3017	2641
amba_decomposed_arbiter_9	9435	18383	19829	16178

the size of the resulting graph. Further analysis was also performed on execution times, to ensure that the algorithm is practically applicable.

After running the original Knor on the examples, it was compared to the sizes of the AIGs with the new conversion method. Getting many different example automata to run Knor on was essential to reaching accurate conclusions. Analysis was further conducted on whether additional changes to the conversion method are conducive or detrimental to our goal.

When inspecting the results from ISOP conversion, it was noticed that large graphs seemed to perform poorly. It was devised that in larger graphs, there was a greater chance for redundancy to take place during conversion. This was due to remaking gates for the same variables, which could simply be reused. An improvement was implemented, by adding a caching mechanism, that checks for every product, if there already exists a gate for any pair of variables in the product. In case of a match, the gate is directly added to the *gate_queue* and the variables are removed from the product. This optimization yielded somewhat more favourable results, as will be explored in the following Evaluation section. The presence of improvements on some graphs indicated that the underlying assumption that redundancy was present in the AIGs was correct.

3.3 Documentation

The documentation of the results was done by writing down the number of AND-gates of the AIGs with both generation methods. The amount of AND-gates is the metric chosen to monitor as it is the principal objective of this research to reduce the size of the AIG. The amounts were then compared as can be seen in the Evaluation section of this document. It contains all the documentation of this research. Additional analysis was carried out to reason about the effectiveness of different conversion methods. Overall statistics, such as the mean and median were calculated. After results were logged, it was reasoned as to why they occur and how they can be improved. This led to two optimization strategies, both of which were documented in the same manner. Finally, a comparison between all the new generation methods was compiled, to give an overview of the results holistically.

4 EVALUATION

4.1 Basic ISOP conversion

The results of implementing this conversion method range between notable improvement and detrimental setback. In general, the algorithm was found to have a wide margin in the degrees of improvement. A detailed documentation of the results can be seen in Table 2. The colours indicate if there has been a reduction or increase in size, with blue and red respectively and grey meaning there was no significant(more than 10%) difference. Knor has a rich library of examples, out of which the table only shows around 10 to avoid cluttering the document. The results are representative of all inputs and were chosen arbitrarily. The file extension of the input Parity Automata formulas file extensions have been omitted for visual clarity, but each of them ends with ".tlsf.ehoa".

In the left column of the table, the example automata used can be seen. The second column shows the size of the output graph in terms of AND-gates for the original ITE conversion. The rightmost column contains the sizes when using the new ISOP conversion, with no optimizations. As can be seen, the new conversion method under-performs on larger examples by generating a more than two times larger AIG. Inspecting these problematic instances indicated there is redundancy in the generated AIG. The redundancy comes from creating new gates when old ones could have been reused. This led to the idea for the following improvement: use of caching. The caching mechanism explained in the Analysis section was implemented.

The amount of AND-gates in the generated graph is roughly equal to the sum of variables in each product and the total amount of products, since one AND-gate is used per variable for conjunction and one gate per product is used for disjunction (along with three

Table 3. Statistics of different conversion methods

Conversion Method	Mean size	Median size	# of graphs reduced	# of graphs unchanged	# of graphs increased
ITE (baseline)	1526	91	0	208	0
ISOP	3942	95	101	10	97
ISOP cached	4564	90	115	9	84
ISOP recursive	3294	80	152	13	43

inverters). Note that the actual number is considerably smaller since caching has been used to avoid remaking gates for the same variables. The relation between these numbers and the number of nodes in the BDD is the underlying cause of the difference in the performance of the two conversion methods for each example.

4.2 ISOP conversion with cache

Without the optimization, the results showed some potential, but it decidedly under-performed on larger examples. The results from the optimization attempt can be seen in Table 2. The fourth column, denoted by ISOPc, shows the results with caching. With the optimization the results became more pronounced, increasing the margins of both the reductions and increases in graph size.
If a gate for two variables already exists and they are both present in a product, it is favourable to use the existing gate, than to arbitrarily pair them up and end up with more gates, but equivalent behaviour. This idea fueled the implementation of this cache optimization. Unfortunately, it didn't yield the expected results, due to reducing the amount of caching that happens in the framework by default. The built-in cache in Knor, upon attempt to make an already existing gate, simply returns that gate. This means that in practice, this optimization has a trade-off, which sometimes ends up making the result worse overall. However, if both caching mechanisms are combined to work synchronously, more significant results might be obtained.

4.3 ISOP recursive conversion

Since the results from using caching were also underwhelming, an entirely different approach was taken to convert the ISOP form into an AIG. Instead of iteratively collecting all the products and creating a two-layer sum of products AIG, a recursive approach was taken, as shown in Algorithm 3. This conversion takes advantage of the structure of the cover returned by the ISOP algorithm. ISOP form is represented by a zero-suppressed binary decision diagram (ZDD)[8]. A ZDD is a BDD in which true edges of nodes cannot point to the *False* leaf of the diagram. This means that if you follow only the high edges you get a product that is a member of the irredundant sum that is represented by the ZDD. The algorithm makes use of that by making AND-gates while following the high edge and "OR"-gates while following the low edge. The results from the low and high edges are recursively obtained.

Algorithm 3 BDD_to_AIG_rec

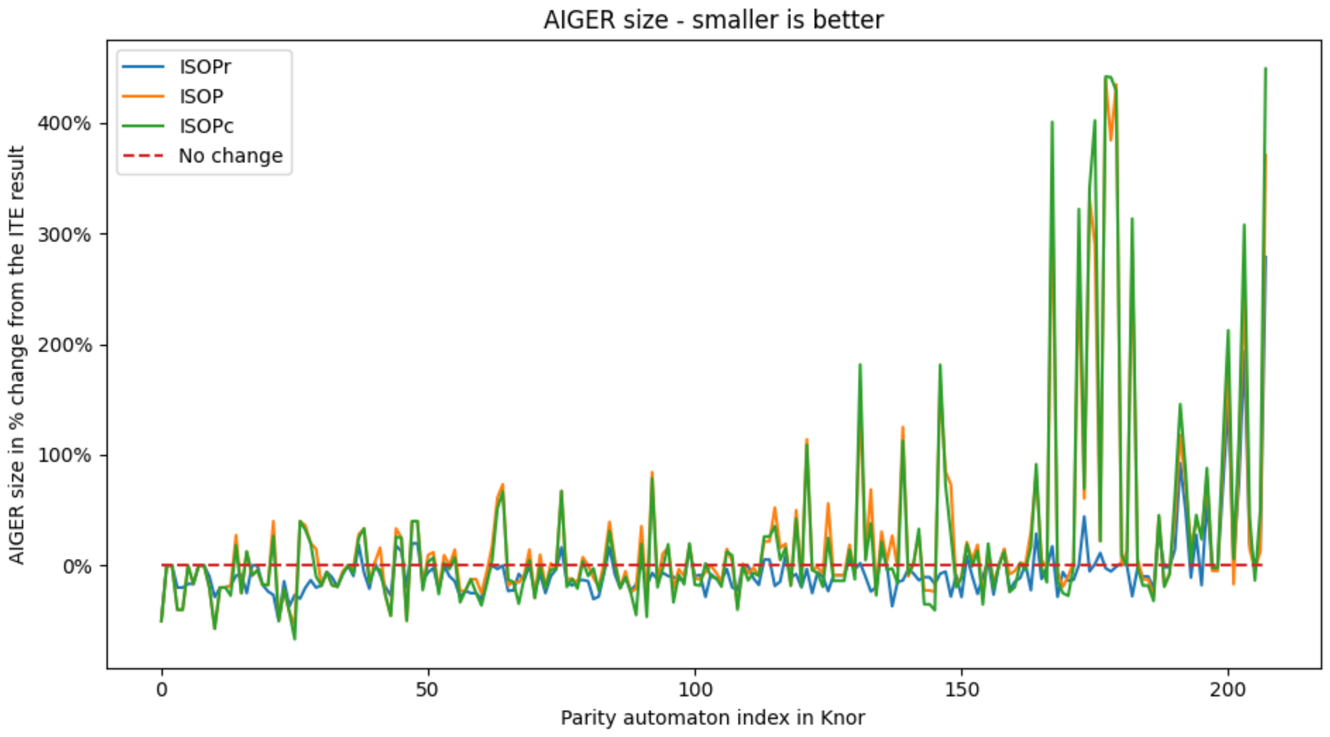
```
$cover \leftarrow isop(BDD)$
**if** $cover == true$ **then return** $aiger\_true$
**else if** $cover == false$ **then return** $aiger\_false$
**end if**
$res \leftarrow zdd\_getvar(cover)$                ▷ Get the variable of the node
$cover\_low \leftarrow zdd\_getlow(cover)$          ▷ The false edge of the cover
$cover\_high \leftarrow zdd\_gethigh(cover)$        ▷ The true edge of the cover
**if** $cover\_low != false$ **then**
    $low\_aig \leftarrow BDD\_to\_AIG\_rec(cover\_low)$
    $res \leftarrow makeand(res, low\_aig)$         ▷ AND-gate for product
**end if**
**if** $cover\_high != true$ **then**
    $high\_aig \leftarrow BDD\_to\_AIG\_rec(cover\_high)$
    $res \leftarrow \neg(makeand(\neg res, \neg high\_aig))$    ▷ "OR"-gate for sum
**end if**
**return** $res$
```

The results of this approach can be seen in Table 2. They can be found in the rightmost column, denoted by ISOPr. The recursive conversion method yields the best results out of all attempted implementations. However, the issue of large examples persists, as it dramatically under-performs compared to the original ITE conversion. The fact that all 3 different ISOP conversions have the same issue suggests that there is an underlying cause for the sub-optimal performance. This will be further addressed in the Discussion section.

4.4 Overall statistics

To obtain a deeper understanding we delve into the more general statistics of the conversion methods. These stats have been noted down in Table 3. The data in question is, for all examples in Knor, the resulting sizes of AIGs generated by each conversion method. The average has blown up with all ISOP conversions, which is owed to large graphs exploding in size. However, the median size of AIGs has been reduced on with both the ISOP cache and the recursive ISOP method. The number of results which have been improved can be seen in the "graphs reduced" column. This allows us to gauge the overall effectiveness of the conversion method. There, we can see that the recursive version of ISOP has a net positive effect on the library of examples.
For further context of the effects of the conversion methods, a plot of the relative change in size for each conversion method can be found in Figure 4. The X-axis of this plot is the index of the example in the Knor repository. The net effects of each conversion method can be seen. The blue line is the recursive implementation of ISOP, which we have concluded to be the best performing, and this plot further reinforces this, as it consistently gives smaller results compared to the other two methods.

Fig. 4. Relative changes of each conversion method compared to ITE, measured in % change from the baseline graph



The plot also showcases the dramatic increase in size for some of the examples. The larger ones are ordered towards the end of the library and as shown in the graph, there are increases of over 400%.

4.5 Execution times

While space and time complexity were not goals for the project, it is important to see how the new conversion method impacts execution speeds, to know if the novel conversion is applicable in practice. It was also expected that the space and time complexity will experience a negative effect, due to the ISOP algorithm having exponential complexity. Memoization was used to tackle this issue and reduce complexity. Fortunately, there is no significant impact on performance. The specific run times can be found in Table 4. Note that the names of the "amba_decomposed" files have been shortened by substituting the "amba_decomposed" with "..." in order to fit everything on one table for the reader's convenience. The table shows the times of each execution time of Knor for every employed method in milliseconds.
These benchmarks were generated by using the "time" command in the Linux terminal on Knor, so it includes parsing and other steps of the process, not just the BDD to AIG conversion. Thus, the changes we observe are more significant percentage-wise for the conversion itself. However, they are nonetheless irrelevant for the practical implications, even for larger examples, being approximately 100 milliseconds worse.

Table 4. Execution times of Knor with different conversion methods

Parity automaton	ms (ITE)	ms (ISOP)	ms (ISOPr)	ms (ISOPc)
ltl2dba_E	3	3	4	4
ltl2dba_R	3	3	3	4
ltl2dba_alpha	3	3	3	3
..._lock	3	3	3	3
ltl2dba_Q	3	3	3	4
ltl2dba_U1	4	3	3	4
ltl2dba01	3	3	3	3
lilydemo22	4	4	4	4
full_arbiter	6	7	6	7
..._tincr	4	4	4	4
..._tsingle	4	4	4	4
..._encode_10	13	15	19	14
SliderDelayed	37	40	40	42
loadfull5	10	11	11	11
..._arbiter	104	114	113	136
..._arbiter_9	1155	1286	1346	1341

5 DISCUSSION

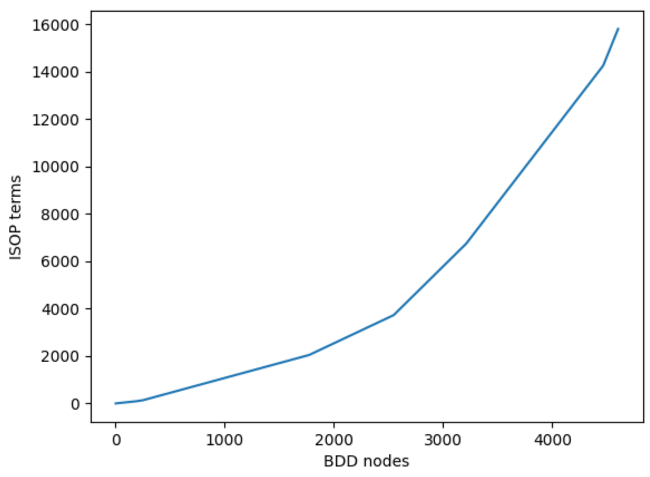
The research aimed to determine if a significant reduction in AIG size could be obtained by using ISOP form of a BDD in a novel conversion method. We also aimed to identify other ways to improve the results. While the results show some promise, an entirely different approach is likely needed to achieve the desired reduction in graph sizes. As all 3 conversion methods had the same weakness, it is safe to assume that, if we want the conversion to work for large examples, we need to explore dramatically different methods for BDD to AIG conversion. But there is yet more that can be done to investigate the ISOP conversion.

An alternative approach to reactive LTL synthesis is employed in the Strix repository[5]. As opposed to the symbolic approach that is proposed by Remco[1], they use explicit state synthesis. It is worthwhile to explore the instances in which this tool produces smaller outputs than Knor, document them and reason as to what is the advantage and how it can be further improved.

5.1 Future Work

As previously mentioned, the difference in sizes between the ITE and ISOP conversions can be linked to the relationship between the size of the input BDD and the size of the ISOP cover of the BDD. For the ITE conversion, the upper bound for the size of the generated AIG is three times the size of the BDD in terms of nodes. For the ISOP conversion, the upper bound is the sum of the amount of terms and products in the cover. Surprisingly, this ratio was unfavourable for large graphs with some BDDs having four times as many ISOP terms as they had nodes. The specific relationship has been plotted in Figure 5.

Fig. 5. Relationship between BDD nodes and ISOP terms for different BDDs



As outlined by Shin-Ichi Minato[7], ISOP forms are neither unique nor minimal. Our generation method may yield inefficiently large covers for sizeable BDDs. It may be worth exploring if a modified ISOP generation algorithm could deliver more minimal covers for larger BDDs.

Furthermore, the cache optimization led to some improvements, which indicates two things. Firstly, it may bring significant results if it is optimized to work with the built-in cache. For example, before removing a pair from a product and reusing a gate, a look-ahead can be done on the overall effect on the size of the AIG. Thus, scenarios, where it bloats the graph, can be avoided. Secondly, this optimization working shows that the assumption that redundancy was present in the graph was indeed correct. This means that an approach where the graph sizes are reduced in a form of post-processing could yield substantially significant results. An algorithm could be devised to determine and remove redundancies in an AIG. A tool that can achieve this in a similar manner already exists - ABC: An Academic Industrial-Strength Verification Tool[3]. Integrating this into Knor might yield substantial improvements.

6 CONCLUSION

Over the course of this research, it has become clear that converting BDDs to AIGs is not a trivial task. The principal objective of reducing the size of generated AIGs was achieved for some graphs, but not for larger instances, where it would have been the most useful. This thesis was not sufficient to identify significant methods to achieve the desired effects, but some potential strategies were outlined. The ISOP method was correctly implemented and it was learnt that, at least with the implementations attempted in this paper, it does not serve the required purpose. The reason for the lack of improvement was identified to be the surprising relationship between the number of BDD nodes and ISOP terms for larger examples. Additional ways to reduce the AIG were identified - study and tweak the ISOP generation so that it performs better on large BDDs; make use of better caching to avoid redundancies in the AIG. Finally, reducing the AIG directly as a post-processing step, as there is evident redundancy in the AIGs.

Overall, in this research we managed to use irredundant prime covers to convert BDDs to AIGs. The results were, however, unsatisfactory in some instances. Nonetheless, we managed to identify the reason for these results and suggested possible improvements and alternative approaches to achieve the principal objective.

SOURCE CODE AVAILABILITY

The work presented in this paper has been merged into the master branches of the Sylvan[12] and Knor[11] repositories, which can be accessed through the references. The results can be reproduced, by compiling and running the Knor[11] repository with and without the "--isop" option. This provides the user with the ability to compare the ITE base conversion to the final ISOP version that was integrated into the repository - the recursive ISOP. We encourage interested educators or researchers to reach out to the author if they have any questions.

ACKNOWLEDGMENTS

The author would like to thank Moritz Hahn for providing insight and explaining some of the abstract concepts of this topic during the early phases of this research. Furthermore, special thanks are in order to the supervisor of this research - Tom van Dijk who played an active role in both the implementation and brainstorming for this thesis. Without his help, this research would not have been fruitful. Finally, the author extends his gratitude to the track chair Peter Lammich, for the helpful comments and feedback provided.

REFERENCES

[1] R. Abraham. 2021. Symbolic LTL Reactive Synthesis. http://essay.utwente.nl/87386/

[2] A. Biere. 2007. The AIGER AndInverter Graph (AIG) format version 20071012. *Institute for Formal Models and Verification* (2007).

[3] Robert Brayton and Alan Mishchenko. 2010. ABC: An Academic Industrial-Strength Verification Tool. In *Computer Aided Verification*, Tayssir Touili, Byron Cook, and Paul Jackson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 24–40.

[4] S. Jacobs and G. A. Perez. 2022. The reactive synthesis competition. http://www.syntcomp.org

[5] Michael Luttenberger, Philipp J. Meyer, and Salomon Sickert. 2020. Practical synthesis of reactive systems from LTL specifications via parity games. *Acta Informatica* 57, 1-2 (2020), 3–36. https://doi.org/10.1007/s00236-019-00349-3

[6] George H. Mealy. 1955. A method for synthesizing sequential circuits. *The Bell System Technical Journal* 34, 5 (1955), 1045–1079. https://doi.org/10.1002/j.1538-7305.1955.tb03788.x

[7] Shin-ichi Minato. 1993. Fast generation of prime-irredundant covers from binary decision diagrams. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* E76-A, 6 (1993), 967–973. www.scopus.com Cited By :25.

[8] Shin-ichi Minato. 1993. Zero-Suppressed BDDs for set manipulation in combinatorial problems. *Proceedings - Design Automation Conference* (1993), 272–277. https://doi.org/10.1145/157485.164890 cited By 516.

[9] Guillermo A. Pérez. 2019. The Extended HOA Format for Synthesis. *CoRR* abs/1912.05793 (2019). arXiv:1912.05793 http://arxiv.org/abs/1912.05793

[10] Fabio Somenzi. 1999. Binary Decision Diagrams. , 303-366 pages. http://www.ecs.umass.edu/ece/labs/vlsicad/ece667/reading/somenzi99bdd.pdf

[11] T. van Dijk. 2020. Knor, a simple synthesis tool for HOA parity automata. https://github.com/trolando/knor

[12] T. van Dijk. 2020. Sylvan - a parallel (multi-core) multi-terminal binary decision diagram library written in C. https://github.com/trolando/Sylvan