Almost-Symbolic Synthesis via
$\Delta_2$-Normalisation for Linear Temporal Logic

Remco Abraham
tv@utwente.nl
University of Twente
Enschede, The Netherlands

Tom van Dijk
t.vandijk@student.utwente.nl
University of Twente
Enschede, The Netherlands

Salomon Sickert
salomon.sickert@mail.huji.ac.il
The Hebrew University of Jerusalem
Jerusalem, Israel

Abstract
The classic approach to synthesis of reactive systems from linear temporal logic (LTL) specifications involves the translation of the specification to a deterministic $\omega$-automaton and computing a winning strategy for the corresponding game with an $\omega$-regular winning condition. Unfortunately, this procedure has an unavoidable double-exponential blow-up in the worst-case and suffers from the state-explosion problem. To address this state-explosion problem in practice we propose an almost-symbolic version of this classic idea that performs the following steps: (1) normalisation of the specification into a Boolean combination of “simple” fragment of LTL, (2) translation of each “simple” subformula into a deterministic automaton, (3) encoding of each automaton into a binary decision diagram (BDD), (4) construction of a parity automaton (and thus game) by operations on the BDD, (5) symbolic computation of a winning strategy, and finally (6) extraction of a symbolic controller. We prototype this approach in the tool Otus, compare it against Strix, the winner of SYNTCOMP 2018-2020, on the SYNTCOMP benchmarks, and identify several specifications where Otus outperforms Strix.

Keywords: Binary Decision Diagram, Linear Temporal Logic, Reactive Synthesis, Parity Game

1 Introduction
In reactive synthesis, as outlined by [18], we are tasked with constructing a controller such that all interactions with an unknown environment satisfy a linear temporal logic (LTL) specifications. It is shown by [18] that this is 2-EXPTIME-complete, and the upper bound is established by constructing a deterministic Rabin automaton (DRW).

Due to the inherent state-space explosion problem, various ideas were developed that make the task more manageable such as: bounded synthesis [9], which tries to find solutions smaller than a fixed bound, or restrictions to less expressive fragments of LTL, e.g., the GR1-fragment [4], which has been successfully been applied in practice [10]. We refer the reader to [2] for in-depth overview.

A few years ago reactive synthesis using deterministic parity automata (DPW, a subclass of DRW) and parity games was deemed infeasible in practice. One reason was the lack of an efficient translation from LTL to deterministic $\omega$-automata, since at that time, translations obtaining deterministic $\omega$-automata used non-deterministic B"{a}bjchi automata (NBW) as an intermediate step and determination of NBW is a famously hard problem. Despite landmark results, such as [20], in practice the deterministic automata quickly became too large. With the rise of direct translations, LTL synthesis tools such as ltlsynt [16] using [19] and Strix [14] using a combination of [7, 8] showed that with clever engineering\(^1\) explicit state-space techniques are capable of solving a wide range of specifications and performed better than some of the previous techniques avoiding DPW. Still, all these techniques need to construct a double-exponentially large state-space in the worst case. We propose a symbolic variant of these algorithms to tackle the state explosion problem. This has been attempted before in [17], where Morgenstern and Schneider propose to use the safety-progress hierarchy developed in [6, 15] to obtain a symbolic reactive synthesis algorithm. This hierarchy shows that every LTL formula is equivalent to a formula from one of six syntactic classes and using the notation from [21] these are denoted $\Sigma_i$, $\Pi_i$, and $\Delta_i$ for $i \in \{1, 2\}$. Furthermore, $\Delta_i$ is the Boolean closure of $\Sigma_i$ and $\Pi_i$. Thus in order to obtain a symbolic algorithm one needs to define a symbolic translation from $\Sigma_i$ and $\Pi_i$. At that time it was unclear how to efficiently compute for an arbitrary formula an equivalent one in the corresponding syntactic classes. Thus [17] also needs to resort to a determination construction. We reexamine this idea, and make use of the new normalisation result from [21] that translates every (future) LTL formula to a formula in $\Delta_2$.

2 Construction
We now detail the six steps of the construction outlined in the beginning. For the rest of the section we fix an LTL formula $\varphi$ over a set of inputs $I$ and outputs $O$.

**Step 1.** The formula $\varphi$ is translated into the $\Delta_2$-normal form using the normalisation procedure by [21]. While it is already established in [6] that every formula can be decomposed into a Boolean combination of persistence ($\Sigma_2$) and recurrence ($\Pi_2$) properties, the constructive proof uses a sub-procedure with non-elementary complexity. [21] addresses this and presents a simple and syntactic translation

----
\(^1\)Strix for example uses on-the-fly exploration of the parity game to avoid constructing the whole game.
to $\Delta_2$ with exponential complexity. Further, this translation
is well-behaved in the sense that one still obtains double-
exponential deterministic Rabin automata (DRW), despite
the exponential blow-up incurred by the normalisation in-
between. Let now $\varphi'$ be a formula in conjunctive normal
form from $\Delta_2$ that is equivalent to $\varphi$.

**Step 2.** Let $\psi_1, \psi_2, \ldots, \psi_n$ be subformulas of $\varphi'$ from $\Pi_2$
and $\Sigma_2$. Each subformula $\psi_i$ is now separately and directly
translated to either a deterministic B\’\^{}ijchii (DBW) or co-
B\’\^{}ijchii automaton (DCW) using the break-point construction
specifically tailored for $\Pi_2$ and $\Sigma_2$ from [21]. The under-
lying assumption is that in practice the intermediate DBW
and DCW are small, since specifications tend to be large
Boolean combinations of small formulas.

**Step 3.** We now switch to a symbolic representation
and encode each DBW and DCW in a binary decision diagram
(BDD). For our prototype we use a simple encoding scheme:
We assign each state of the automaton an integer and we use
the binary representation in the BDD. Thus if an automaton
has $n$ states, we use $\lceil \log_2(n) \rceil$ variables to encode a state.
Thus storing the transition relation of a single automaton
requires $2 \cdot \lceil \log_2(n) \rceil + |I| + |O| + 1$ variables. The last variable
is used to store if an edge is either accepting or rejecting.
Observe that there might be encoding schemes that yield
smaller BDDs and we leave an investigation of the effect of
better encoding schemes as future work.

In this prototype we use a simple, fixed categorical vari-
able ordering as follows: Atomic propositions are at the top
of the BDD. Next come variables encoding the current state,
then variables encoding the acceptance condition, and fi-
nally variables encoding the successor state. We leave evalu-
itng the impact of other fixed orderings or dynamic vari-
able reordering to future work.

**Step 4.** We obtain a symbolically represented DRW from
Step 3 by union and intersection following the structure of
$\varphi'$. We implement union and intersection of automata by
“and”-operations in the underlying BDD to obtain the prod-
uct automaton. In order to obtain a deterministic parity au-
tomaton (DPW) we apply a symbolic implementation of the
“typeness”-construction from [5]. For this we implement the
symbolic SCC-decomposition due to [3] as a sub-procedure.
We rely on two results from [5]: (1) Given a DRW $R$ one can
effectively compute if there exists a parity acceptance con-
dition $\gamma$ on the structure of $R$ such that the resulting parity
automaton accepts the same language. (2) Let $R$ and $S$ be
a DRW and deterministic Streett automaton (DSW) for the
same language $L$. Then the algorithm in (1) always finds a
parity acceptance condition $\gamma$ on top of the product automa-
ton $R \times S$ where we ignore the acceptance condition of $S$. We
use this construction in the following way: We translate $\varphi$
into a DRW $R$ and apply (1). If this succeeds, we continue to
Step 5 with the obtained DPW $P$. Otherwise, we construct a
DSW $S$ by translating $\neg \varphi$ to a DRW and then complement-
ing the acceptance condition. We then build the symbolic
product automaton $R \times S$ using an “and”-operation in the
underlying BDD, apply (1) again and obtain a DPW $P$.

**Step 5.** The symbolic DPW is reinterpreted as a parity
game and we apply the distraction fix-point iteration [23]
to compute a winning strategy for either the environment
or the system. This algorithm has been shown to be com-
petitive and easy to implement symbolically [13].

**Step 6.** The BDD representing the winning strategy for
the system player (if there is one) is then converted to an
and-inverter graph (AIG) with Mealy semantics. We resolve
potential non-determinism in the symbolic representation,
i.e., “don’t cares”, by preferring to output 0 for don’t cares.
We leave a refinement of this simple heuristic as future work,
e.g., one could try to find an assignment for the output val-
ues that yield the smallest BDD representing the winning
strategy.

3 Experimental Evaluation

We implement the proposed approach in the tool Otus and
base it on the LTL and $\omega$-automata library Owl [12] and
on the multi-core BDD library Sylvan [22]. We evaluate the
construction using a subset of the specifications of the SYNT-
COMP competition [11] and compare it against Strix (ver-
sion 2020.06-Syntcomp) using the configuration\footnote{We use strix -f "$formula" --ins "$ins" --outs "$outs"
--no-compress-circuit --auto -e pq -c.} that was
ranked at the first place in SYNTCOMP 2020. Since we only
measure the time needed for synthesis and not the quality,
the size, of the circuits, we disable post-processing using
ABC’s AIGER minimization tool [1] for Strix.

The experiments are run on a cluster of Dell PowerEdge
M610 servers with two Xeon E5520 processors and each run
gets 8 cores and 56 GB memory assigned.

In total we use 421 realizable and 157 unrealizable speci-
fications from github.com/meyerphi/syntcomp-reference and
filter the specifications as follows: for each specification a
five minute time-budget is allocated and only specifications
that are processed within five minutes by each tool are se-
lected. We thus select 320 realizable and 85 unrealizable
specifications. For selected formulas we repeat the experiment
five times and collect the average execution times. The re-
results are presented in Figure 1 and 2 for the realizable
and unrealizable specifications, respectively.

We observe that the results are mixed. This is to be ex-
pected, since the goal of our prototype, which skips several
possible optimisations, is to evaluate the feasibility of our
approach and not develop a tool that outperforms Strix.
However, we identify specifications that are challenging for
Strix, but can be solved comparably fast by Otus. Indeed,
for some specifications our construction is over 30x faster.
Almost-Symbolic Synthesis via Normalisation for Linear Temporal Logic

Figure 1. Execution time comparison of Otus (vertical) vs. Strix (horizontal) for realizable specifications; different magnification levels.

Figure 2. Execution time comparison of Otus (vertical) vs. Strix (horizontal) for unrealizable specifications; different magnification levels.

This indicates that the symbolic technique is promising to tackle larger specifications in the future. For many specifications, Strix is over 1000× faster than our construction, but these are often inputs that Strix can solve in just a few seconds. One reason for this is that Strix has several optimizations, including early termination that avoid computing the full parity game, which we currently do not do. Further, the symbolic technique we propose has a certain fixed overhead that potentially amortises with larger specifications. We list in Table 1 ten instances, with highest and lowest ratios in runtime. These are specifications that can be solved by both our construction and by Strix.

4 Conclusion

We outlined an “almost”-symbolic algorithm for LTL synthesis using parity games, implemented a prototype, and compared it against the explicit-state, on-the-fly LTL synthesis tool Strix. Although step (2) uses an explicit representation and in the subsequent steps we often choose the simple and naive approaches, e.g., state encoding with a simple fixed variable ordering, we observe promising results for a subset of the specifications. This motivates further research into the construction and refinement of each of the outlined steps. Finally, we expect that we can also reduce the execution time by better engineering of the tool, e.g., reducing the number of variables used in the BDDs.

Table 1. The ten specifications with the highest and lowest total execution time ratio of the construction over Strix. Execution times are presented in seconds and are rounded to two decimals. Ratios are computed using the exact execution times.

<table>
<thead>
<tr>
<th>Specification</th>
<th>Otus (s)</th>
<th>Strix (s)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>collector_v1_5</td>
<td>0.72</td>
<td>24.02</td>
<td>33.19956</td>
</tr>
<tr>
<td>amba_decomposed_lock_10</td>
<td>1.86</td>
<td>37.59</td>
<td>20.15616</td>
</tr>
<tr>
<td>amba_decomposed_lock_6</td>
<td>0.33</td>
<td>4.24</td>
<td>12.77229</td>
</tr>
<tr>
<td>LedMatrix</td>
<td>22.15</td>
<td>224.09</td>
<td>10.11631</td>
</tr>
<tr>
<td>amba_decomposed_encode_10</td>
<td>1.28</td>
<td>9.74</td>
<td>7.62835</td>
</tr>
<tr>
<td>TwoCounters3</td>
<td>0.67</td>
<td>4.38</td>
<td>6.50238</td>
</tr>
<tr>
<td>lt12dba_beta_5</td>
<td>2.66</td>
<td>14.62</td>
<td>5.48810</td>
</tr>
<tr>
<td>lilydemo22</td>
<td>0.24</td>
<td>1.26</td>
<td>5.32375</td>
</tr>
<tr>
<td>tictactoe</td>
<td>2.31</td>
<td>12.16</td>
<td>5.27006</td>
</tr>
<tr>
<td>amba_decomposed_lock_8</td>
<td>0.54</td>
<td>2.54</td>
<td>4.66618</td>
</tr>
<tr>
<td>lt12dba_C1_6</td>
<td>27.72</td>
<td>0.11</td>
<td>0.00399</td>
</tr>
<tr>
<td>EscalatorSmart</td>
<td>226.97</td>
<td>0.91</td>
<td>0.00399</td>
</tr>
<tr>
<td>escalator_smart</td>
<td>232.94</td>
<td>0.90</td>
<td>0.00386</td>
</tr>
<tr>
<td>prioritized_arbiter_6</td>
<td>109.24</td>
<td>0.42</td>
<td>0.00385</td>
</tr>
<tr>
<td>lt12dba_C2_5</td>
<td>26.29</td>
<td>0.10</td>
<td>0.00366</td>
</tr>
<tr>
<td>detector_5</td>
<td>26.27</td>
<td>0.09</td>
<td>0.00346</td>
</tr>
<tr>
<td>lt12dba_p03</td>
<td>81.31</td>
<td>0.16</td>
<td>0.00195</td>
</tr>
<tr>
<td>lt12dba_C1_7</td>
<td>176.54</td>
<td>0.12</td>
<td>0.00066</td>
</tr>
<tr>
<td>lt12dba_C2_6</td>
<td>176.26</td>
<td>0.11</td>
<td>0.00064</td>
</tr>
<tr>
<td>detector_6</td>
<td>176.77</td>
<td>0.10</td>
<td>0.00059</td>
</tr>
</tbody>
</table>

Acknowledgments

The authors want to thank Orna Kupferman for suggesting to use [5] for the symbolic translation of deterministic Rabin automata to deterministic parity automata. Tom van Dijk is supported by the European UnionÄ‘s Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No 893732. Salomon Sickert is supported by the Deutsche Forschungsgemeinschaft (DFG) under project number (436811179).

References


