# Knor: reactive synthesis using Oink

Tom van Dijk(✉), Feije van Abbema, and Naum Tomov

Formal Methods and Tools
University of Twente, Enschede, The Netherlands
`t.vandijk@utwente.nl`, `{f.vanabbema,n.tomov}@student.utwente.nl`

**Abstract.** We present an innovative approach to the reactive synthesis of parity automaton specifications, which plays a pivotal role in the synthesis of linear temporal logic. We find that our method efficiently solves the SYNTCOMP synthesis competition benchmarks for parity automata from LTL specifications, solving all 288 models in under a minute. We therefore direct our attention to optimizing the circuit size and propose several methods to reduce the size of the constructed circuits: (1) leveraging different parity game solvers, (2) applying bisimulation minimisation to the winning strategy, (3) using alternative encodings from the strategy to an and-inverter graph, (4) integrating post-processing with the ABC tool. We implement these methods in the Knor tool, which has secured us multiple victories in the PGAME track of the SYNTCOMP competition.

**Keywords:** Reactive synthesis · Parity games · Binary decision diagrams

## 1 Introduction

Reactive synthesis as first stated by Church [8,9] and outlined in [32] is the act of automatically constructing a reactive system such that all interactions with an unknown environment satisfy a linear temporal logic (LTL) specification. While early solutions were proposed to solve the synthesis problem via finite-state automata [7], until recently reactive synthesis using deterministic parity automata and parity games was deemed infeasible in practice, in part due to the lack of efficient translations from LTL to deterministic $\omega$-automata. With the rise of direct translations, LTL synthesis tools such as ltlsynt [27,33,34] and Strix [26] are capable of solving a wide range of specifications via deterministic parity automata and parity games, and perform better than some of the previous techniques avoiding deterministic parity automata.

The advantage of reactive synthesis is that synthesized systems are correct by construction and therefore do not need to be tested nor model checked for correctness. The reactive synthesis (SYNTCOMP) competition was founded to increase the impact of reactive synthesis in industry and improve the quality of synthesis tools [22,23]. Motivated by the new PGAME track in the SYNTCOMP competition, we seek to use the Oink parity game solver [11] in the competition and to implement the necessary infrastructure that translates the parity automata of the competition into parity games suitable for Oink, and that translates the winning strategy computed by Oink into a Boolean circuit. We name this implementation **Knor**[1].

Knor leverages Oink to solve parity games with state-of-the-art parity game solvers [16], and the Sylvan binary decision diagrams (BDD) package [14] to implement most of the steps before and after solving and a purely symbolic parity game solver based on [25]. The techniques implemented in Knor have secured us multiple victories in the SYNTCOMP competition, in 2021, 2022 and 2023.

Following initial success of Knor in the competition, we observe a major difference with main competitors ltlsynt and Strix. While Knor can solve all benchmarks in a remarkably short time, the constructed circuits are sometimes several orders of magnitude larger than the circuits constructed by other tools. Thus, we propose several techniques, mostly symbolic techniques that rely on binary decision diagrams, to reduce the size of the constructed circuits.

*Contribution.* We present the Knor tool that solves the synthesis problem of parity automata to Boolean circuits, built around the parity game solver Oink. We consider three methods to translate the given parity automaton to a parity game, and present a novel symbolic approach that improves upon an explicit translation by several orders of magnitude. As Oink implements several parity game solvers that have been shown in [16] to perform well for parity games derived from reactive synthesis benchmarks, we consider whether changing the algorithm impacts the size of the constructed circuit. We study whether applying bisimulation minimisation as in [15], which aims to minimize the number of states of the winning strategy after solving the parity game, can reduce the size of the circuits. Similarly, we study different encodings from the winning strategies into Boolean logic, in particular whether a onehot encoding of the states improves the circuit size. Finally, we apply a similar post-processing step as Strix by using the ABC tool [4,5] to minimize the constructed circuit after encoding it as an and-inverter graph. Sec. 3 describes Knor and provides accessible descriptions of the implemented techniques. We evaluate these techniques in Sec. 4. We discuss our findings in Sec. 5.

## 2 Preliminaries

Given two disjoint sets of Boolean variables $I$ and $O$ representing input and output signals, and an $\omega$-regular language $L$ of infinite words over the alphabet $2^{I \cup O}$ representing a specification, the reactive synthesis problem asks us to construct a controller that enforces $L$. The controller is a function $\left(2^{I \cup O}\right)^* \times 2^I \to 2^O$ that yields a valuation of the output signals $2^O$ based on a history of input and output signals $\left(2^{I \cup O}\right)^*$ and the current input signals $2^I$.

While we are interested in the broader context of the synthesis of reactive systems that enforce specifications given in linear temporal logic (LTL), we

[1] Knor is the Dutch word for the sound that a pig makes, i.e., “oink”.

assume in this paper that $L$ is given as a deterministic parity automaton. LTL specifications can be translated to a parity automaton of doubly-exponential size.

Deterministic parity automata (DPA) are $\omega$-regular automata that accept $\omega$-regular languages. A DPA is a tuple $(Q, q_0, AP, \Delta, F)$, where $Q$ is a finite set of states, $q_0 \in Q$ is the initial state, $AP$ is a set of atomic propositions, $\Delta \subseteq Q \times 2^{AP} \times Q$ is the transition relation and $F\colon Q \to \mathbb{N}$ assigns to each state a *priority*. A *run* of the automaton is an infinite sequence of states consistent with the transition relation. A run is accepting if and only if the maximum priority that occurs infinitely often along the run is an even number. We define parity automata with priorities on states. Alternatively, priorities can also be on transitions.

A parity game is a DPA with two players Even and Odd, where the set of states $Q$ is partitioned into two sets $Q_0$ and $Q_1$. In this paper, we refer to the states of the parity game as vertices and the transitions of the parity game as edges. A run on a parity game is an infinite sequence of vertices where player Even decides the next vertex if the current vertex is in $Q_0$, and player Odd if it is in $Q_1$. A fundamental result for parity games is that they are memoryless determined [18], i.e., each vertex is winning for exactly one player, and both players have a positional strategy for each of their winning vertices.

To solve the synthesis problem, given a deterministic parity automaton over $AP = I \cup O$, we construct a parity game by *splitting* the automaton across $I$ and $O$, letting one player (the environment) choose a valuation of variables in $I$ and the other player (the controller) a valuation of variables in $O$.

The result of reactive synthesis is a Boolean circuit, structured as an and-inverter graph (AIG). An AIG is a directed acyclic graph, featuring terminal nodes that denote Boolean inputs (input signals and latches), internal nodes representing AND-gates, and edges with complementation for logical negation.

Binary decision diagrams [6,17] (BDDs) are a well known data structure for representing and manipulating Boolean functions. A binary decision diagram is a rooted, directed acyclic graph. Its internal nodes represent decisions based on the values of Boolean variables, directing the path to one of the two child nodes, via the "true" edge (depicted as a solid arrow) and the "false" edge (depicted as a dashed arrow). Reaching the terminal node "1" indicates that the represented Boolean function evaluates to true for that particular valuation, and reaching the "0" node indicates a false evaluation. BDDs are recognized as a canonical representation of Boolean functions when they meet two conditions. First, they must be ordered; that is, they follow a fixed variable ordering when encountering Boolean variables. Second, they must be reduced, meaning that any redundant decision nodes with identical successors are eliminated [6]. BDDs can be incredibly efficient if a suitable variable ordering is found and the represented set is encoded in a way that results in small decision diagrams.

Multi-terminal binary decision diagrams (MTBDDs) extend BDDs by allowing terminal nodes to hold various types of data, not just the Boolean values true and false. The MTBDD implementation in Sylvan [14] in particular allows for terminal nodes to be labeled by 64-bit values. These labels can represent a wide

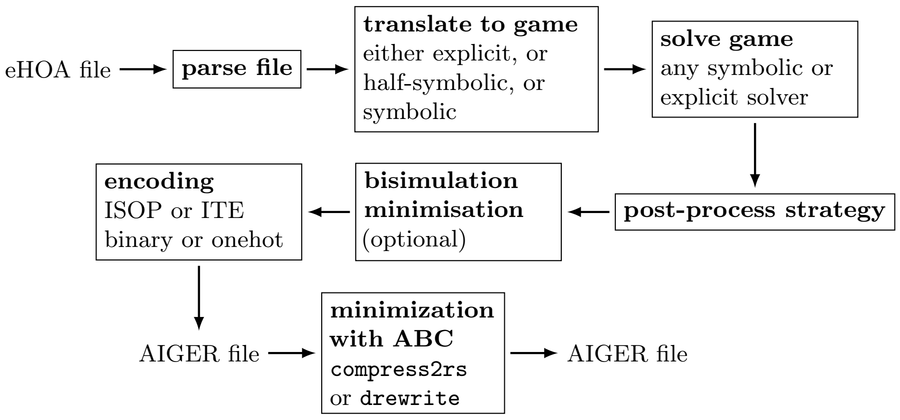

**Fig. 1.** Overview of Knor from input file to output file.

range of data, including 64-bit integers, pointers, floating-point numbers, or even pairs of 32-bit values.

## 3 Knor

We study reactive synthesis from parity automata to Boolean circuits in the **Knor** research tool. Knor is written in C++ and is publicly available under a permissive license via https://www.github.com/trolando/knor. See Fig. 1 for an overview of Knor. All steps of the program are discussed in the following sections.

### 3.1 Input format

Knor reads input files formatted using the extended Hanoi Omega-Automata (HOA) format [31].

The HOA format [1] is a file format to describe finite-state automata that accept sets of infinite words. The automata consist of a finite set of states $Q$, one or more initial states $I \subseteq Q$, a set of atomic propositions $AP$, and a labeled transition relation $\Delta \subseteq Q \times \mathbb{B}(AP) \times Q$, where each transition is labeled with a Boolean formula $\phi \in \mathbb{B}(AP)$, where we use $\mathbb{B}(AP)$ to denote the set of Boolean formulas over $AP$. Furthermore, the HOA format describes an acceptance condition of the automaton, i.e., a set of infinite runs of the automaton which are considered accepting. For the purposes of the current paper, we are only interested in the *parity condition*, i.e., the automaton is accepting if and only if the lowest/highest priority seen infinitely often along the run is even/odd, depending on whether the acceptance condition is `min even`, `min odd`, `max even` or `max odd`. In the HOA format, the priorities are either on states or on transitions.

The extended HOA format adds a distinction between controllable (output) and uncontrollable (input) atomic propositions [31].

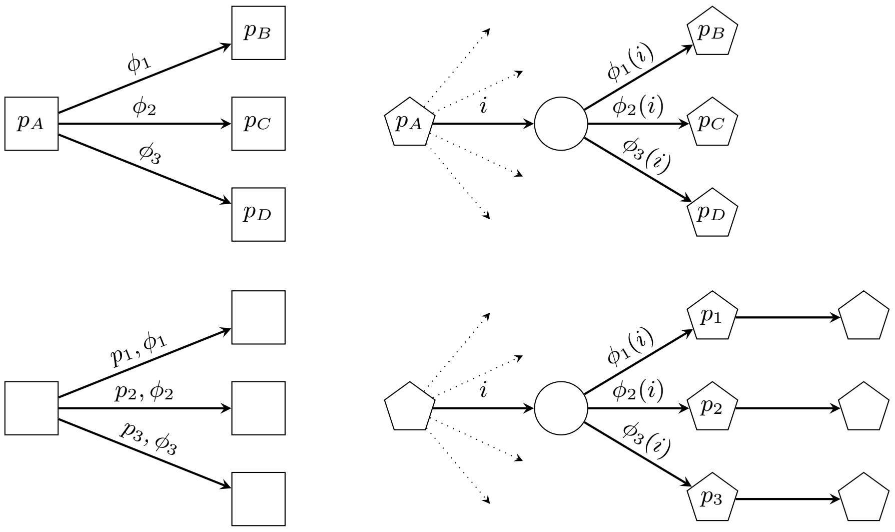

**Fig. 2.** Splitting a transition on the parity automaton (left) to construct the parity game (right), with priorities on the states (above) or on the transitions (below). We depict states by squares, vertices of the environment player by pentagons and vertices of the controller player by circles.

## 3.2 Output format

Knor can produce parity games in the standard PGSolver [20] format that is also accepted by Oink, as well as Boolean circuits in the AIGER format [3].

## 3.3 Translation from automaton to game

As described above, the parity automaton consists of a number of states with transitions labeled by a Boolean formula, and with the priorities either on the transitions or on the states.

To translate the automaton to a parity game, we need to split every transition into two parts. The environment player "moves first" by choosing a valuation of the input signals, and the controller player responds by setting output signals such that the specification is guaranteed. That is, the output signals are determined by the current state and the current input signals.

We propose three methods to convert the parity automaton to a parity game: a naive explicit method, a half-symbolic method and a fully symbolic method.

**(Naive) Explicit method.** The explicit method simply creates a parity game vertex for every state in the parity automaton, and then splits the transitions into two parts as in Fig. 2.

For every valuation $i$ of the input signals, we create an intermediate vertex that is controlled by the controller player. This intermediate vertex should have the least relevant priority, typically 0. For every transition with a label (Boolean formula) that is satisfiable for $i$, we then create an edge from the intermediate vertex to the successor of the transition.

Since we want our parity games to have priorities on the vertices and not on the edges, we need to create extra vertices in case the automaton has priorities on transitions. This is also shown in Fig. 2. Priorities on the source vertex, intermediate vertex, and target vertices should be set to the least relevant priority (typically 0) or be ignored by the solver.

The result is an explicit parity game which Knor directly constructs using Oink. The game is then solved with any algorithm implemented by Oink.

**Half-symbolic method.** The fully explicit method works reasonably well for many of the smaller input models, however some models result in a significant exponential blowup of the parity game, as any game with $n$ input signals has $2^n$ outgoing edges per source vertex. The extended HOA format actually encodes the labels on the transitions *symbolically* using Boolean formulas, so an exponential blowup in some cases can be expected. We propose a method that still results in an explicit game constructed using Oink, but that employs binary decision diagrams to reduce the number of intermediate vertices and extra transitions in the parity game.

For every state, we produce a multi-terminal binary decision diagram (MTBDD) encoding all outgoing transitions, with decision variables representing input signals ordered before variables representing output signals, and terminal nodes encoding both priority and successor state as a pair of two 32-bit numbers.

We then collect all **subroots** of the MTBDD after the input signals, i.e., along each path from the root node to a terminal node, we find the first node that is either a decision node with a variable of an output signal, or a terminal node. For every such node $N$, we create a corresponding intermediate vertex owned by the controller player. The paths leading to $N$ correspond to valuations of the input signals that lead to that intermediate vertex, where the controller can decide how to respond. We let the controller choose to go to any state (vertex) encoded by a terminal node that is reachable from $N$. For every such terminal node, we simply add an edge from the intermediate vertex to the target vertex.

**Fully-symbolic method.** While the half-symbolic method already results in a major reduction in the size of the parity games, we can go further and encode the full transition relation of the parity automaton as a single BDD, which can then automatically be interpreted as a symbolic parity game simply by ordering variables as follows:

1. Variables $s$ corresponding to the source state.
2. Variables $i$ corresponding to input signals.
3. Variables $o$ corresponding to output signals.

4. Variables $p$ and $s'$ corresponding to the priority (either from the transition or from the target state) and the target state.

One can read this BDD intuitively as follows: given some current state (1) and some current input values (2), if the controller sets certain output values (3) we arrive with some priority at our next state (4). Variables within these four groups can be ordered freely; however, we implement a naive approach and have not optimized this ordering; this is left as an opportunity for future work.

Since we encode the entire automaton as a single BDD, states that share some transitions can benefit from the automatic reduction offered by BDDs.

We present a translation from this symbolic parity game to an explicit parity game that explicitly uses the structure of the decision diagram to construct the game. This procedure consists of the following steps:

1. We create a **state vertex** controlled by the environment player for every state (with transitions) in the symbolic parity game. These vertices get priority 0.
2. Along each path in the BDD, we find the first decision node *after* the input signals. We create an **intermediate vertex** controlled by the controller player for every such node. These vertices also get priority 0.
3. Along each path in the BDD, we find the first decision node *after* the output signals. We decode the priority and the target state and create a **priority vertex** for the environment player with the decoded priority and with a single edge to the state vertex corresponding to the target state.
4. For every state, we compute the reachable decision nodes of step 2 and create edges from the state vertices to the intermediate vertices.
5. For every decision node of step 2, we compute the reachable decision nodes of step 3 and create edges from the intermediate vertices to the priority vertices.

Further improvements to this procedure are possible by considering that vertices may share many transitions, and additional vertices could be added based on the structure of the BDD. This could reduce the number of edges at the cost of more vertices. Furthermore, we do not merge the state vertices and priority vertices, which might reduce the number of vertices. This is left as an opportunity for future work.

## 3.4 Solving the parity game

Using the procedure described above, we can produce an explicit parity game that can be solved by Oink. As shown in [16], several solvers implemented in Oink are very efficient for parity games derived from reactive synthesis:

- strategy iteration (`psi`) [11,19]
- tangle learning (`tl`) [10]
- priority promotion (`npp`) [2,11]
- Zielonka's recursive algorithm (`zlk`) [11,35]
- fixpoint iteration using freezing (`fpi`) [16]
- fixpoint iteration using justifications (`fpj`) [24]

We also implement a symbolic solver based on [25]. This symbolic solver implements fixpoint iteration with freezing using BDD operations, and operates directly on the BDD obtained by the fully-symbolic translation.

### 3.5 Post-processing the strategy

After applying the strategy to the symbolic parity game, we perform two post-processing steps. In the case that the strategy does not give all output signals a value, we default to setting output signals to false (or 0). We also compute all reachable vertices of the parity game from the initial state vertex, restricted to the winning strategy, and remove unreachable vertices.

### 3.6 Bisimulation minimisation

To further reduce the number of vertices of the parity game, we apply bisimulation minimisation. Bisimulation minimisation computes equivalence classes of vertices, i.e., all vertices that have the same behavior w.r.t. input and output signals. We use the signature-based partition refinement approach of [15].

Recall that the symbolic parity game is a BDD over the variables $s, i, o, p, s'$ as described in Sec. 3.3. We first drop the priority variables $p$ from the BDD, as the priorities on the states are not relevant after solving. We reserve fresh BDD variables $c$ for the classes, which are ordered *after* the next state variables, i.e., $s < i < o < s' < c$. We maintain the current assignment from states to classes in a BDD over variables $s'$ and $c$. The reason for $s'$ rather than $s$ is that this reduces the number of BDD operations. The initial partition assigns all states to a single equivalence class. We then repeatedly compute the current signature of all states, which is a BDD encoding for every state the *classes* that can be reached and the input/output values to reach them, as follows:

1. Given a BDD $G$ encoding the symbolic parity game over the variables $s, i, o, s'$, and a BDD $P$ encoding the current partition over the variables $s'$ and $c$, we compute the BDD $S$ representing the signatures over variables $s, i, o, c$ by performing the operation `and_exists`$(G, P, s')$.
2. We use the `refine` operation of [15] to replace the signatures (over variables $i, o, c$) in $S$ by new classes, reusing previous class identifiers whenever possible, and renaming $s$ variables to $s'$ variables on-the-fly, resulting in the next BDD $P$ over the variables $s'$ and $c$.
3. We repeat steps 1 and 2 until the number of classes is stable.

Afterwards, we apply the obtained partition by replacing the states in the symbolic parity game by the equivalence classes.

### 3.7 Encoding the strategy as a circuit

There are several methods to create a Boolean circuit from the solver parity game. We first need to encode all reachable states of the parity game as latches in the

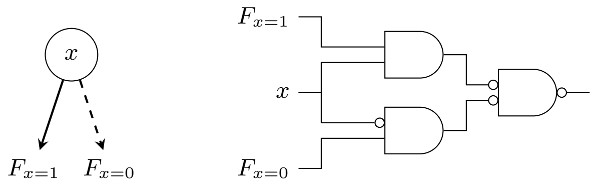

**Fig. 3.** Sketch of the encoding from a BDD decision node (left) to three AND-gates (right), representing the Boolean formula $(F_{x=1} \wedge x) \vee (\neg x \wedge F_{x=0})$.

Boolean circuit. We employ two methods for this: (1) one latch per state; and (2) one latch per BDD state variable. We call the former method `onehot` and the latter `binary`; in the first case at all times only a single latch is set, whereas in the second case the latches form a binary encoding of the states, similar to how they are encoded in the symbolic parity game. As the initial state of a Boolean circuit has all latches reset (to 0), we invert the latch that encodes the initial state for the `onehot` encoding and we encode the initial state as state 0 for the `binary` encoding.

We then compute a BDD $F$ for every latch and for every output signal, where $F$ is a BDD over the variables $s, i$ (current state and current input signals) such that the latch or signal will be set if and only if $F$ evaluates to true. We then translate each BDD $F$ to an and-inverter graph. Again we propose two methods to achieve this:

– by using Shannon expansion (`ITE`) as in Fig. 3 recursively;
– by first obtaining the irredundant sum-of-products [28] (`ISOP`) of $F$ in the form of a ZBDD [29], which can then directly be translated to an AIG: first all products are created, and then the products are connected through inverted AND-gates (as $ab \vee cd \equiv \neg(\neg(ab) \wedge \neg(cd))$).

We thus have four combinations: `ITE` with `binary` or `onehot` encoding and `ISOP` with `binary` or `onehot` encoding. Furthermore, we use a cache when creating AND-gates to avoid duplicate gates.

## 3.8 Post-processing with ABC

After encoding the strategy as a circuit, we apply optional post-processing of the circuit using ABC [5].

Similar to Strix, we apply the `compress2rs` script, which is described in [4]. The `compress2rs` script performs rewriting, refactoring, balancing, and truth-table-based resubstitution. While Strix applies the script until no further improvement is found, we halt when the improvement is less than 2.5%.

We also apply a sequence of three ABC commands, `drw`, `balance` and `drf`, which we call the `drewrite` script here. We apply this script until the improvement is less than 1%.

### 3.9 Usage of Knor

Knor expects an eHOA file on standard input; it also accepts a filename as a command line parameter instead. With the options `-a` and `-b`, Knor writes the constructed circuits to standard output as an AIGER file in ASCII or binary format respectively. With the option `-v`, Knor prints timings and other information to standard error.

By default, Knor uses the fully symbolic translation to a parity game. One can use `--naive` for the naive explicit encoding and `--explicit` for the half-symbolic encoding, and `--print-game` to print the resulting parity game in PGSolver format to standard output. Only the fully symbolic translation supports the full synthesis pipeline.

To choose an explicit-state solver of Oink, one can pick any solver from the list obtained with `--solvers`, in particular the solvers `--tl`, `--npp`, `--fpi`, `--fpj`, `--psi`. and `--zlk`. To solve using the symbolic solver, use `--sym`. With the option `--real`, Knor will only decide realizability and use tangle learning (`--tl`) as the default solver. The default solver for synthesis is the symbolic solver (`--sym`).

Bisimulation minimisation is applied by default, unless the `--no-bisim` option is used. To encode the circuit, Knor uses by default ITE and onehot encoding. To change this one can use the options `--isop` and `--binary`. To apply post-processing with ABC after constructing the circuits, use the options `--compress` and `--drewrite`.

## 4 Empirical Evaluation

We present the empirical results here.

### 4.1 Benchmarking

We evaluate the techniques implemented in Knor using the benchmarks of SYNTCOMP for the PGAME track that come from reactive synthesis, i.e., they are based on LTL specifications in the TLSF file format. In recent years, SYNTCOMP has also incorporated benchmarks in the PGAME track that do not come from reactive synthesis, such as artificial hard games that are designed to be time consuming for specific parity game solvers. Oink can easily handle such hard games by using a solver for which no hard game has been designed yet, and since our aim is to develop techniques for reactive synthesis specifically, we limit ourselves to benchmarks from the TLSF dataset[2]. We also exclude input files that are not parity automata; this removes the `aut*.ehoa` files, two `test*.ehoa` files, and `UnderapproxStrengthenedDemo`, which is a Büchi automaton consisting of a single state. In total 288 input files remain.

The benchmarks are run on a machine with an Intel i5-13600KF processor. This is a 14-core processor, but we only use a single thread. Knor is compiled using gcc version 13.2.1. We repeat benchmarks 5 times and take the median to obtain

[2] https://github.com/SYNTCOMP/benchmarks/tree/v2023.4/parity/tlsf_based

| **Model** | **explicit** | **half-symbolic** | **symbolic** |
|---|---:|---:|---:|
| `amba_decomposed_lock_15` | T.O. | 46 | 24 |
| `amba_decomposed_lock_14` | T.O. | 46 | 24 |
| `amba_decomposed_lock_13` | T.O. | 46 | 24 |
| `TwoCountersDisButA9` | T.O. | 668,065 | 7,249 |
| `amba_decomposed_lock_12` | 402,997,254 | 46 | 24 |
| `amba_decomposed_lock_11` | 100,820,998 | 46 | 24 |
| `amba_decomposed_lock_10` | 25,237,510 | 46 | 24 |
| `TwoCountersGui` | 21,022,475 | 256 | 155 |
| `TwoCountersDisButA8` | 15,254,863 | 497,310 | 4,721 |
| `full_arbiter_8` | 11,287,306 | 1,669,066 | 177,690 |
| `amba_decomposed_lock_9` | 6,323,718 | 46 | 24 |
| `amba_decomposed_encode_16` | 4,981,507 | 876 | 330 |
| `TwoCountersDisButA7` | 3,939,305 | 98,947 | 2,365 |
| `TwoCountersDisButA6` | 3,806,249 | 101,175 | 1,733 |

**Table 1.** Sizes in number of vertices of the largest parity games, sorted descending by size of parity games constructed using the explicit method.

| **Technique** | **Sum of Vertices** | **Time** (sec) |
|---|---:|---:|
| explicit | 622,987,565 | 1,177.91 |
| half-symbolic | 8,491,540 | 18.28 |
| symbolic | 620,510 | 11.76 |

**Table 2.** Cumulative size of parity games and time required for construction of the parity games of the 284 inputs that could be constructed by all three techniques.

the runtimes. All experimental scripts and log files are available as [12], and are also available online via http://www.github.com/trolando/knor-experiments.

## 4.2 Translating the parity automaton to a parity game

We first compare the three different techniques to obtain a parity game from the parity automaton: **explicit**, **half-symbolic** (only symbolic splitting) and **fully symbolic**.

Of the 288 benchmarks, the explicit method could not construct the parity game for four benchmarks within the timeout of 3600 seconds. See Table 1 for the largest parity games constructed by the explicit method, as well as the four input models for which no parity game could be constructed within 3600 seconds. The two other methods could construct the parity games within a reasonable amount of time, as is displayed in Table 2. The given time is only the time required for constructing the games and excludes time required for parsing the input file, which is the same for all methods.

Clearly, the fully symbolic method is superior to the other methods, both in the speed of construction and in the size of the constructed parity games. When

| **Solver** | **Circuit size** binary | onehot | **Time** (sec) |
|---|---|---|---|
| symbolic fpi (`--sym`) | 317,403 | 122,514 | 18.45 |
| fixpoint with justifications (`--fpj`) | 350,035 | 139,900 | 0.16 |
| fixpoint with freezing (`--fpi`) | 353,120 | 140,297 | 0.22 |
| strategy iteration (`--psi`) | 334,149 | 140,916 | 0.57 |
| priority promotion (`--npp`) | 427,048 | 161,244 | 0.17 |
| Zielonka (`--zlk`) | 480,472 | 175,427 | 0.18 |
| tangle learning (`--tl`) | 604,044 | 213,632 | 0.17 |

**Table 3.** Cumulative circuit size in number of gates and cumulative solving time in number of seconds for the tested parity game solvers.

we consider individual input models, we find 20 cases where the half-symbolic approach results in slightly smaller parity games than the fully symbolic approach. The largest difference is 13 vertices (100 vertices instead of 113 vertices), which is negligible compared to the several orders of magnitude advantage that the fully symbolic method has in larger parity games, as Table 1 demonstrates. The cumulative time for the fully symbolic method is dominated by a handful of input models that require more than a second. Almost all parity games are constructed in fewer than 10 milliseconds.

Although the size of the parity game does not necessarily always correspond to the size of the constructed circuit or the required time for the entire synthesis process, it seems an obvious choice to only consider the fully symbolic translation in the remainder of this study.

## 4.3 Solving the parity game

We consider several parity game solvers, which have been shown in the past to be successful for solving games derived from synthesis: Zielonka's recursive algorithm, priority promotion, tangle learning, the two fixpoint algorithms using freezing and justifications, strategy iteration, and symbolic fixpoint iteration. One of these, symbolic fixpoint iteration, directly operates on the symbolic parity game constructed by the fully symbolic method. All other solvers require the procedure outlined in Sec. 3.3 to translate the symbolic representation to an explicit game. The game is then solved, and we construct the circuit using the standard ITE encoding and either the binary or the onehot encoding of the states. We do not yet perform bisimulation minimisation or postprocessing using ABC.

The reason that it is interesting to consider different solvers is that different solvers may result in entirely different strategies to win the parity game. In particular, it may be that some solvers favor winning regions that reach either higher priorities or lower priorities, which can result in significant differences. This is in fact supported by the results presented here.

We report runtimes **for solving the parity games** (thus excluding time before solving and after solving) as well as the sizes of the circuits in Table 3.

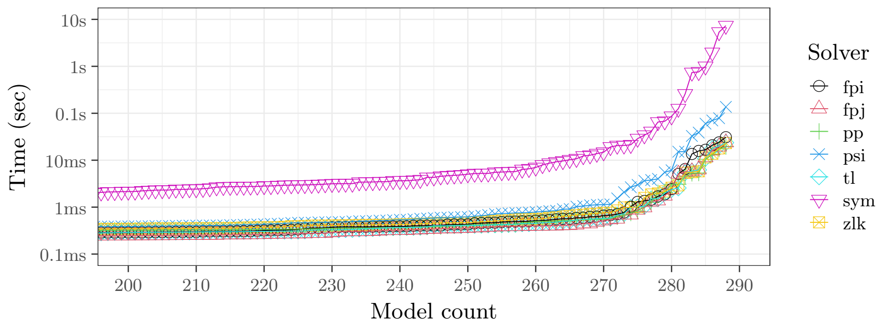

**Fig. 4.** Cactus plot of the number of parity games that can be solved within the given amount of time per solver.

| **Model** | **tl** | **sym** | **pp** | **psi** | **zlk** | **fpi** | **fpj** |
|---|---|---|---|---|---|---|---|
| `generalized_buffer_unreal1` | 0.02 | 7.36 | 0.02 | 0.14 | 0.02 | 0.03 | 0.02 |
| `generalized_buffer` | 0.01 | 5.37 | 0.01 | 0.07 | 0.01 | 0.02 | 0.01 |
| `genbuf2` | 0.01 | 1.98 | 0.01 | 0.03 | 0.01 | 0.01 | 0.01 |
| `full_arbiter_unreal3` | 0.00 | 1.00 | 0.00 | 0.06 | 0.00 | 0.02 | 0.01 |
| `amba_decomposed_arbiter_10` | 0.02 | 0.76 | 0.01 | 0.04 | 0.01 | 0.02 | 0.02 |
| `full_arbiter_8` | 0.02 | 0.74 | 0.02 | 0.08 | 0.02 | 0.02 | 0.02 |

**Table 4.** Overview of individual runtimes of each solver in seconds for the benchmarks for which at least one solver requires at least 500 milliseconds.

We observe that only the symbolic algorithm requires any time at all. The other algorithms each require less than a second to solve *all* benchmarks! When we consider the circuit sizes, the fully symbolic algorithm is superior with a cumulative 122,514 gates for all circuits. If we are interested in the best solver that solves all benchmarks in a fraction of a second, then clearly FPJ is the best algorithm, with a cumulative time of 0.16 seconds and a cumulative circuit size of 139,900 gates, although the difference with FPI is not that great.

*Remarks.* The solving time with the symbolic fixpoint iteration algorithm is dominated by just a few benchmarks. All algorithms solve the vast majority of parity games in a fraction of a second. See Fig. 4. Notice the logarithmic scale and that the vast majority of models are computed within a second for all solvers. Just a few models require more than 500 milliseconds to be solved, as is shown in Table 4.

We also did not take parallel operation into account. The symbolic FPI solver, the explicit FPI solver, and the strategy iteration solver have parallel implementations; the symbolic solver leverages the automatic parallelisation of decision diagram operations in Sylvan.

| **Solver** | **Circuit size** | | **Time** (sec) |
|---|---|---|---|
| | binary | onehot | |
| symbolic fpi (`--sym`) + minimisation | 166,839 | 106,500 | 0.19 |
| fixpoint with justifications (`--fpj`) + min. | 205,937 | 124,489 | 0.15 |
| symbolic fpi (`--sym`) | 317,403 | 122,514 | – |
| fixpoint with justifications (`--fpj`) | 350,035 | 139,900 | – |

**Table 5.** Cumulative circuit size in number of gates and cumulative minimisation time in number of seconds for the symbolic fpi and the fixpoint with justifications solvers, with and without bisimulation minimisation after solving.

| **Solver** | **Encoding** | **Circuit size** | **Time** |
|---|---|---|---|
| symbolic fpi (`--sym`) | ISOP, onehot | 102,294 | 0.69 |
| symbolic fpi (`--sym`) | ITE, onehot | 106,500 | 0.61 |
| fixpoint with justifications (`--fpj`) | ISOP, onehot | 113,134 | 0.72 |
| fixpoint with justifications (`--fpj`) | ITE, onehot | 124,489 | 0.64 |
| symbolic fpi (`--sym`) | ITE, binary | 166,839 | 0.09 |
| fixpoint with justifications (`--fpj`) | ITE, binary | 205,937 | 0.12 |
| symbolic fpi (`--sym`) | ISOP, binary | 431,316 | 1.39 |
| fixpoint with justifications (`--fpj`) | ISOP, binary | 476,502 | 1.61 |

**Table 6.** Cumulative circuit size in number of gates and cumulative encoding time in seconds for the symbolic fpi and fixpoint with justification solvers, after bisimulation minimisation, using different encodings to obtain the circuit.

### 4.4 Bisimulation minimisation

We study the effects of bisimulation minimisation for the fully symbolic fixpoint iteration solver and for the explicit fixpoint iteration with justifications solver implemented in Oink.

As Table 5 shows, running bisimulation minimisation on the resulting strategy reduces the total circuit size in all cases. The required time to perform bisimulation minimisation is negligible with a cumulative time of a fraction of a second.

Bisimulation minimisation does not always improve the circuit size. There are a few cases where the procedure slightly increases the circuit size. There are also several models where the circuit size is reduced by several orders of magnitude. Interestingly, in some cases the circuit size is reduced to 0 AND-gates. It seems worthwhile to always apply bisimulation minimisation.

### 4.5 Encoding strategy to circuit

We now consider different encodings from the BDD of the strategy to the controller circuit. See Table 6. Surprisingly, the combination of ISOP and a binary encoding leads to a significantly worse result; whereas using ISOP with a onehot encoding slightly reduces the circuit sizes, but not by a significant amount.

| Solver | Encoding | Method | Circuit size | Time |
|---|---|---|---|---|
| symbolic fpi (`--sym`) | ISOP | compress | 61,434 | 149.26 |
| symbolic fpi (`--sym`) | ITE | compress | 62,506 | 121.27 |
| fixpoint with justifications (`--fpj`) | ISOP | compress | 71,240 | 125.29 |
| fixpoint with justifications (`--fpj`) | ITE | compress | 72,897 | 108.10 |
| symbolic fpi (`--sym`) | ISOP | drewrite | 80,077 | 58.72 |
| symbolic fpi (`--sym`) | ITE | drewrite | 80,425 | 53.21 |
| fixpoint with justifications (`--fpj`) | ISOP | drewrite | 80,454 | 60.88 |
| fixpoint with justifications (`--fpj`) | ITE | drewrite | 80,903 | 58.58 |
| symbolic fpi (`--sym`) | ISOP | | 102,294 | 44.88 |
| symbolic fpi (`--sym`) | ITE | | 106,500 | 39.81 |
| fixpoint with justifications (`--fpj`) | ISOP | | 113,134 | 31.66 |
| fixpoint with justifications (`--fpj`) | ITE | | 124,489 | 25.77 |

**Table 7.** Cumulative circuit size in number of gates for the two solvers, after bisimulation minimisation and using onehot encoding, then using different postprocessing methods to reduce circuit sizes. Given times are **total times** from parsing until writing, in seconds.

| Tool | Circuit size | |
|---|---|---|
| | no post-processing | with post-processing |
| strix | 68,550 | 41,314 |
| sym-bisim-isop-onehot | 87,823 | 50,624 |
| ltlsynt | 544,804 | 98,996 |

**Table 8.** Cumulative size of the circuits for the 201 realizable inputs that could be constructed by all three tools, before and after post-processing with ABC.

Looking at individual benchmarks, we find that the most interesting differences occur with the `full_arbiter_*` and `amba_decomposed_arbiter_*` benchmarks. For these benchmarks, ISOP performs much worse than ITE with a binary encoding, but shows moderate improvement with the onehot encoding.

While there are some differences in the encoding times between the different approaches, the cumulative encoding time is less than two seconds in all cases.

## 4.6 Postprocessing with ABC

Finally, we apply postprocessing of the constructed circuit using ABC. See Table 7 for the results. We observe a very clear tradeoff of space and time. The best result is obtained by using the `compress` algorithm, which reduces the number of gates by about 40%, but this triples the runtime.

## 4.7 Comparison with other tools

We compare Knor to the tools Strix [26] and ltlsynt [27,33,34]. We obtain the two competing tools from the SYNTCOMP 2023 artifact [21]. We use the following command lines, similar to those used in the SYNTCOMP 2023 competition, to run the tools:

- Run Strix without post-processing in ABC:
  `strix --auto --no-compress-circuit -t --hoa <filename>`
- Run Strix with post-processing in ABC:
  `strix --auto -t --hoa <filename>`
- Run ltlsynt (without post-processing in ABC):
  `ltlsynt --from-pgame=<filename> --aiger --verbose`

In the competition, ltlsynt had optional post-processing in ABC as part of the script rather than the executable. This script executed the following ABC commands: `collapse;strash;refactor;rewrite`. The Strix executable runs an embedded version of ABC, repeating the `compress2rs` script until no more improvement is found. To improve the fairness of the comparison, we change the post-processing for ltlsynt to start with `collapse;strash`, as this re-encoding of the circuit via binary decision diagrams significantly improves upon the circuit encoding by ltlsynt, followed by repeating the `compress2rs` script until there is no more improvement. This gives better results than obtained by ltlsynt in SYNTCOMP 2023.

Only 208 of the 288 input files are realizable. Of these, Strix did not solve the following inputs within the 3600 seconds time limit: `amba_decomposed_lock_14`, `amba_decomposed_lock_15`, `Automata325`, `Gamelogic`, `genbuf2`, `SPIPureNext`, `generalized_buffer`. Except for `amba_decomposed_lock_15`, ltlsynt solved all inputs. Disregarding inputs that could not be solved by Strix or ltlsynt, we have 201 realizable inputs that can be solved within the time limit by all three tools. We provide the results with and without post-processing using ABC in Table 8. Considering individual results, we observe that Strix yielded smaller circuits in 142 cases (147 with post-processing) and Knor yielded smaller circuits in 47 cases (also 47 with post-processing). For the larger circuits, the `amba_decomposed_arbiter_*` inputs favored Knor (1527 vs 8282 gates, after post-processing), while Strix did better on the `full_arbiter_` inputs (1594 vs 26040 gates, after post-processing).

Table 8 clearly shows that all tools benefit from the post-processing. While Strix gives the best results for circuit size, the cumulative circuit size of Knor is only 23% more. Knor solves the entire set of inputs, including post-processing by ABC, in about 2.5 minutes, while Strix and ltlsynt cannot solve some benchmarks within the time limit of 1 hour, before post-processing.

## 5 Discussion

In this work, we studied techniques to improve reactive synthesis of parity automata to Boolean circuits using a new tool named **Knor**. We proposed a number of techniques and empirically evaluated these techniques using the benchmarks of the SYNTCOMP competition derived from LTL specifications. Knor has won the PGAME track of the competition several times.

The evidence presented in the empirical evaluation suggests that the best approach for deciding **realizability** is to use the fully symbolic translation from parity automaton to parity game, and any fast explicit-state parity game solver (like a tangle learning variation) for which no hard games have yet been designed. The latter is only needed to counteract any efforts aimed at impairing Knor's performance in SYNTCOMP through the introduction of artificially difficult benchmarks.

For **synthesis**, considering a low circuit size as our primary objective, the clear solution is to use either symbolic fpi (`--sym`) or fixpoint with justifications (`--fpj`), preferring the former at the cost of speed in a few benchmarks, always apply bisimulation minimisation (`--bisim`), use a onehot encoding (`--onehot`) with either ITE or ISOP encoding, and apply postprocessing using ABC's `compress2rs` script (`--compress`).

Knor is publicly available via https://www.github.com/trolando/knor.

**Future work** There are many opportunities for future improvements to the entire pipeline. We already mentioned playing with the variable ordering within the variable groups of the symbolic parity game, and considering slightly more efficient translations from the symbolic parity game to an explicit game in Oink.

We could also consider designing a parity game solving algorithm that explicitly results in small strategies. Some solvers might yield a multi-strategy, where multiple edges in the parity game can be taken to win the game. This could potentially be exploited to simplify the circuits.

It may also be useful to consider bisimulation minimisation on the parity game before solving, and to change the encoding of the states into the BDD, as we currently use a naive binary encoding of the state identifiers in the eHOA format. There may also be other encoding strategies to obtain the Boolean circuit, such as a different encoding of the latches or the approach of [30].

Beyond the reactive synthesis of parity automaton specifications, we may also explore symbolic techniques, including those outlined in this paper, for the synthesis of LTL specifications, building on the preliminary results from our earlier prototype described in [13].

## Acknowledgements

We thank Alan Mishchenko for his helpful comments on using ABC for Boolean circuit minimisation. The first author is supported by the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No 893732.

## Data availability statement

The software, benchmarks and analysed dataset are available as [12]. In addition, the version of Knor studied in the current paper is tagged in the Github repository of Knor as: https://github.com/trolando/knor/tree/TACAS24.

## References

1. Babiak, T., Blahoudek, F., Duret-Lutz, A., Klein, J., Kretínský, J., Müller, D., Parker, D., Strejcek, J.: The Hanoi Omega-Automata Format. In: CAV (1). Lecture Notes in Computer Science, vol. 9206, pp. 479–486. Springer (2015)
2. Benerecetti, M., Dell'Erba, D., Mogavero, F.: Solving Parity Games via Priority Promotion. In: CAV 2016. LNCS, vol. 9780, pp. 270–290. Springer (2016)
3. Biere, A., Heljanko, K., Wieringa, S.: AIGER 1.9 and beyond. Tech. Rep. 11/2, Formal Models and Verification, Johannes Kepler University (2011), https://fmv.jku.at/papers/BiereHeljankoWieringa-FMV-TR-11-2.pdf
4. Brayton, R., Mishchenko, A.: Scalable logic synthesis using a simple circuit structure. In: Proc. of Internal Workshop on Logic Synthesis. vol. 6, pp. 15–22 (2006)
5. Brayton, R.K., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: CAV. Lecture Notes in Computer Science, vol. 6174, pp. 24–40. Springer (2010)
6. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv. **24**(3), 293–318 (1992)
7. Buchi, J.R., Landweber, L.H.: Solving sequential conditions by finite-state strategies. Transactions of the American Mathematical Society **138**, 295–311 (1969)
8. Church, A.: Application of recursive arithmetic to the problem of circuit synthesis. Summaries of the Summer Institute of Symbolic Logic **1**, 3–50 (1957)
9. Church, A.: Logic, arithmetic, and automata. In: Proceedings of the International Congress of Mathematicians. pp. 23–35 (1962)
10. van Dijk, T.: Attracting tangles to solve parity games. In: CAV (2). Lecture Notes in Computer Science, vol. 10982, pp. 198–215. Springer (2018)
11. van Dijk, T.: Oink: An implementation and evaluation of modern parity game solvers. In: TACAS (1). Lecture Notes in Computer Science, vol. 10805, pp. 291–308. Springer (2018)
12. van Dijk, T.: Artifact of Knor: reactive synthesis using Oink (2023). https://doi.org/10.4121/8794d8c0-5959-42f9-ba34-68f2137145a7
13. van Dijk, T., Abraham, R., Sickert, S.: Almost-symbolic synthesis via delta-2-normalisation for linear temporal logic. In: 10th Workshop on Synthesis (2021)
14. van Dijk, T., van de Pol, J.: Sylvan: multi-core framework for decision diagrams. Int. J. Softw. Tools Technol. Transf. **19**(6), 675–696 (2017)
15. van Dijk, T., van de Pol, J.: Multi-core symbolic bisimulation minimisation. Int. J. Softw. Tools Technol. Transf. **20**(2), 157–177 (2018)
16. van Dijk, T., Rubbens, B.: Simple fixpoint iteration to solve parity games. In: GandALF. EPTCS, vol. 305, pp. 123–139 (2019)
17. Drechsler, R., Sieling, D.: Binary decision diagrams in theory and practice. Int. J. Softw. Tools Technol. Transf. **3**(2), 112–136 (2001)
18. Emerson, E.A., Jutla, C.S.: Tree automata, mu-calculus and determinacy (extended abstract). In: FOCS. pp. 368–377. IEEE Computer Society (1991)
19. Fearnley, J.: Efficient parallel strategy improvement for parity games. In: CAV (2). LNCS, vol. 10427, pp. 137–154. Springer (2017)
20. Friedmann, O., Lange, M.: Solving parity games in practice. In: ATVA. LNCS, vol. 5799, pp. 182–196. Springer (2009)
21. Jacobs, S., Perez, G., Schlehuber-Caissier, P.: Data, scripts, and results from SYNTCOMP 2023. Zenodo (2023). https://doi.org/10.5281/zenodo.8161423
22. Jacobs, S., Bloem, R.: The reactive synthesis competition: SYNTCOMP 2016 and beyond. In: SYNT@CAV. EPTCS, vol. 229, pp. 133–148 (2016)

23. Jacobs, S., Pérez, G.A., Abraham, R., Bruyère, V., Cadilhac, M., Colange, M., Delfosse, C., van Dijk, T., Duret-Lutz, A., Faymonville, P., Finkbeiner, B., Khalimov, A., Klein, F., Luttenberger, M., Meyer, K.J., Michaud, T., Pommellet, A., Renkin, F., Schlehuber-Caissier, P., Sakr, M., Sickert, S., Staquet, G., Tamines, C., Tentrup, L., Walker, A.: The reactive synthesis competition (SYNTCOMP): 2018-2021. CoRR **abs/2206.00251** (2022)
24. Lapauw, R., Bruynooghe, M., Denecker, M.: Improving parity game solvers with justifications. In: VMCAI. Lecture Notes in Computer Science, vol. 11990, pp. 449–470. Springer (2020)
25. Lijzenga, O., van Dijk, T.: Symbolic parity game solvers that yield winning strategies. In: GandALF. EPTCS, vol. 326, pp. 18–32 (2020)
26. Luttenberger, M., Meyer, P.J., Sickert, S.: Practical synthesis of reactive systems from LTL specifications via parity games. Acta Informatica **57**(1-2), 3–36 (2020)
27. Michaud, T., Colange, M.: Reactive synthesis from ltl specification with spot. In: Proceedings of the 7th Workshop on Synthesis, SYNT@CAV. vol. 5 (2018)
28. Minato, S.: Fast generation of prime-irredundant covers from binary decision diagrams. IEICE transactions on fundamentals of electronics, communications and computer sciences **76**(6), 967–973 (1993)
29. Minato, S.: Zero-suppressed bdds for set manipulation in combinatorial problems. In: DAC. pp. 272–277. ACM Press (1993)
30. Miyasaka, Y., Mishchenko, A., Wawrzynek, J., Fraser, N.J.: Synthesizing a class of practical boolean functions using truth tables. In: 31st International Workshop on Logic and Synthesis (2022)
31. Pérez, G.A.: The extended HOA format for synthesis. CoRR **abs/1912.05793** (2019)
32. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL. pp. 179–190. ACM Press (1989)
33. Renkin, F., Schlehuber, P., Duret-Lutz, A., Pommellet, A.: Improvements to ltlsynt. In: 10th Workshop on Synthesis (2021)
34. Renkin, F., Schlehuber-Caissier, P., Duret-Lutz, A., Pommellet, A.: Dissecting ltlsynt. Formal Methods in System Design (2023). https://doi.org/10.1007/s10703-022-00407-6
35. Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. Theor. Comput. Sci. **200**(1-2), 135–183 (1998)