International Journal on Software Tools for Technology Transfer (2024) 26:551–567
https://doi.org/10.1007/s10009-024-00754-1

COMPETITIONS AND CHALLENGES

Regular

The Reactive Synthesis Competition (SYNTCOMP): 2018–2021

Swen Jacobs[1] · Guillermo A. Pérez[2] · Remco Abraham[3] · Véronique Bruyère[4] · Michaël Cadilhac[5] · Maximilien Colange[6] · Charly Delfosse[4] · Tom van Dijk[3] · Alexandre Duret-Lutz[6] · Peter Faymonville[7] · Bernd Finkbeiner[1] · Ayrat Khalimov[8] · Felix Klein[7] · Michael Luttenberger[9] · Klara Meyer[9] · Thibaud Michaud[6] · Adrien Pommellet[6] · Florian Renkin[6] · Philipp Schlehuber-Caissier[6] · Mouhammad Sakr[10] · Salomon Sickert[9] · Gaëtan Staquet[4] · Clément Tamines[4] · Leander Tentrup[7] · Adam Walker[11]

Accepted: 17 May 2024 / Published online: 11 June 2024

Abstract

We report on the last four editions of the reactive synthesis competition (SYNTCOMP 2018–2021). We briefly describe the evaluation scheme and the experimental setup of SYNTCOMP. Then we introduce new benchmark classes that have been added to the SYNTCOMP library and give an overview of the participants of SYNTCOMP. Finally, we present and analyze the results of our experimental evaluations, including a ranking of tools with respect to quantity and quality—that is, the total size in terms of logic and memory elements—of solutions.

Keywords Reactive synthesis · Algorithmic verification · Zero-sum games

1 Introduction

Reactive systems are systems that maintain a continuous interaction with their environment. The act of automatically constructing such a system from a given formal specification (or determining that no such system exists) is called *reactive synthesis*. The current definition of reactive synthesis is usually attributed to Alonzo Church (Church [19, 20]). In the last 60 years, several works have laid the theoretical foundations that underpin all current synthesis algorithms for different instantiations of reactive synthesis. Indeed, depending on the format in which a specification for the reactive system is formalized, various synthesis problems arise. For instance, the competition currently has three such specification formats: one for safety specifications defined by monitoring circuits, one for linear-temporal-logic specifications, and one for parity-game specifications.

Reactive synthesis has the potential to revolutionize the way in which reactive systems are designed. This is due to the fact that a synthesized system is correct by construction and therefore does not need to be tested nor verified for correctness. Despite its potential, industry has not yet adopted it, nor the prototype tools are implemented by academic researchers. This is in contrast to other formal verification techniques such as model checking (Baier and Katoen [5], Clarke et al. [21]). With an aim at increasing the impact of reactive synthesis in industry and improving the quality of synthesis tools, the reactive synthesis competition (SYNTCOMP) was founded in 2014 (Jacobs et al. [41]). In short, the competition is designed to foster research into well-engineered, scalable, and user-friendly synthesis tools. To realize this, the competition organizers have proposed standards for benchmark formats and maintain a library of benchmarks with entries that

✉ S. Jacobs
jacobs@cispa.de

✉ G.A. Pérez
guillermo.perez@uantwerp.be

1 CISPA Helmholtz Center for Information Security, Stuhlsatzenhaus 5, Saarbrücken, 66123, Germany

2 University of Antwerp – Flanders Make, Middelheimlaan 1, Antwerpen, 2020, Belgium

3 University of Twente, Enschede, The Netherlands

4 University of Mons, Mons, Belgium

5 DePaul University, Chicago, USA

6 EPITA Research Laboratory, Kremlin-Bicêtre, France

7 Saarland University, Saarbrücken, Germany

8 Université libre de Bruxelles, Brussels, Belgium

9 Technische Universität München, Germany

10 University of Luxembourg, Luxembourg, Luxembourg

11 Independent Researcher, Sydney, Australia

remain challenging for state-of-the-art tools. Most importantly, SYNTCOMP provides a dedicated and independent platform for the objective comparison of synthesis tools.

SYNTCOMP has become an annual event associated with the International Conference on Computer Aided Verification (CAV) and the Workshop on Synthesis (SYNT). The organizational team of the competition has changed slightly from its inception: In 2019, Guillermo A. Pérez joined the organization team, and since 2020, the competition has an advisory committee that presently consists of Roderick Bloem, Armin Biere, Salomon Sickert, Jean-François Raskin, Bernd Finkbeiner, and Ayrat Khalimov. Every year the organizers publish a call for solvers and benchmarks on the website[1] and via the associated mailing list.[2]

In this paper, we present the list of benchmark families that have been added to the competition from 2018 to 2021 and the tools that participated in the competition during the same years. Finally, we also highlight the most interesting experimental results from these editions of the competition and discuss the progress of synthesis tools observed in this time.

2 Setup, rules, and execution

We begin this section with a reminder on the foundations of the synthesis problem. For more detail on reactive synthesis and (parity-)game solving, we refer the reader to the corresponding chapters in the books of Clarke et al. [21] and Apt and Grädel [2].

2.1 Synthesis and realizability

To model the execution of a reactive system, we make use of infinite sequences: An *infinite word* α over an *alphabet* A is a function $\alpha\colon \mathbb{N}_{>0} \to A$. Thus we write $\alpha(i)$ to refer to the ith letter of α. We denote by A^ω the set of all infinite words over A.

Definition 1 (Infinite-word automata)
An (ω-word) *automaton* is a tuple $\mathcal{N} = (Q, q_0, A, \Delta)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, A is a finite alphabet, and $\Delta \subseteq Q \times A \times Q$ is the transition relation. We assume that for all $p \in Q$ and $a \in A$, there exists $q \in Q$ such that $(p, a, q) \in Q$.

The automaton is said to be *deterministic* if for all $p \in Q$ and $a \in A$, we have that $(p, a, q_1), (p, a, q_2) \in \Delta$ implies $q_1 = q_2$. A *run* of $\mathcal{N}$ on a word $\alpha \in A^\omega$ is an infinite sequence $\rho = q_0\alpha(1)q_1\alpha(2)\cdots \in (Q \cdot A)^\omega$ such that $(q_i, \alpha(i+1), q_{i+1}) \in \Delta$ for all $i \in \mathbb{N}$.

Automata are paired with a condition that determines which runs are *accepting*. For the competition, we consider four such *acceptance conditions*.

- The *safety* condition is defined with respect to a set $U \subseteq Q$ of *unsafe states*. A run $\rho = q_0a_1q_1a_2\ldots$ is accepting for this condition, or *safe*, if and only if for all $i \in \mathbb{N}$, we have that $q_i \notin U$.
- The *Büchi* condition is defined with respect to a set of *Büchi* states $B \subseteq Q$. A run $\rho = q_0a_1q_1a_2\ldots$ is accepting for this condition if and only if for all $i \in \mathbb{N}$, there exists $j \geq i$ such that $q_j \in B$, that is, the run visits Büchi states infinitely often.
- The *co-Büchi* condition is also defined with respect to a set $B \subseteq Q$ of states, which are in this case *rejecting* states. A run $\rho = q_0a_1q_1a_2\ldots$ is accepting for the condition if and only if there exists $i \in \mathbb{N}$ such that $q_j \notin B$ for all $j \geq i$, that is, rejecting states are visited only finitely often.
- The *parity* condition is defined with respect to a *priority function* $p\colon Q \to \mathbb{N}$. A run $\rho = q_0a_1q_1a_2\ldots$ is accepting for the parity condition if and only if the value $\liminf_{i\to\infty} p(q_i)$ is even, that is, the smallest priority that appears infinitely often along the run is even.

A word α is accepted by an automaton $\mathcal{N}$ if it has a run on α that is accepting. In the sequel, it will sometimes be useful to consider *universal automata*. Such an automaton $\mathcal{N}$ accepts a word α if all its runs on α are accepting.

We denote by $\mathcal{L}(\mathcal{N})$ the *language* of the automaton $\mathcal{N}$, that is, the set of words that $\mathcal{N}$ accepts.

2.1.1 Synthesis games and strategies

Definition 2 (Games)
A *(Gale–Stewart) game* on input and output alphabets I and O, respectively, is a two-player perfect-information game played by Eve and Adam in rounds: Adam chooses an element $i_k \in I$, and Eve responds with an element $o_k \in O$. A *play* in such a game is an infinite word $\langle i_1, o_1\rangle\langle i_2, o_2\rangle \cdots \in (I \times O)^\omega$.

A game is paired with a *payoff set* $P \subseteq (I \times O)^\omega$ that determines who wins a play π. If $\pi \in P$, then π *is winning for Eve*; otherwise, it is *winning for Adam*.

Definition 3 (Strategies)
A *strategy* for Adam is a function $\tau\colon (I \times O)^* \to I$ that maps every (possibly empty) play prefix to a choice of input letter. Similarly, a *strategy for Eve* is a function $\sigma\colon (I \times O)^* I \to O$ that maps every play prefix and input letter to a choice of output letter.

[1] http://www.syntcomp.org/.

[2] https://lists.iaik.tugraz.at/cgi-bin/mailman/listinfo/syntcomp.

A play $\pi = \langle i_1, o_1\rangle\langle i_2, o_2\rangle \dots$ is *consistent with* a strategy τ for Adam if $i_k = \tau(\langle i_1, o_1\rangle \dots \langle i_{k-1}, o_{k-1}\rangle)$ for all $k \in \mathbb{N}$; it is consistent with a strategy σ for Eve if $o_k = \sigma(\langle i_1, o_1\rangle \dots \langle i_{k-1}, o_{k-1}\rangle, i_k)$. A pair of strategies σ and τ for Eve and Adam, respectively, induces a unique play $\pi_{\sigma\tau}$ consistent with both σ and τ.

In a game with payoff set P, the strategy σ for Eve is a *winning strategy* if for all strategies τ for Adam, we have $\pi_{\sigma\tau} \in P$; the strategy τ for Adam is winning if for all strategies σ for Eve, we have $\pi_{\sigma\tau} \notin P$.

The realizability and synthesis problems are defined for games whose payoff sets are given as the language of an automaton. We sometimes refer to these as *games played on automata*.

Definition 4 (Realizability and synthesis)
Consider finite input and output alphabets I and O, respectively, and an automaton $\mathcal{N}$ with alphabet $I \times O$. The *realizability problem* asks whether there exists a winning strategy for Eve in the game with payoff set $\mathcal{L}(\mathcal{N})$. The *synthesis problem* further asks to compute and output such a strategy if one exists.

In SYNTCOMP, solvers are asked to solve the realizability and synthesis problems. More precisely, the competition is organized into separate tracks according to different specification formats for games (see Sect. 2.2), and in each track the participants are asked to solve these two problems.

2.1.2 Finite-memory strategies

A *finite-memory* strategy σ for Eve in a game played on the automaton $\mathcal{N} = (Q, q_0, I \times O, \Delta)$ with finite input and output alphabets I and O is a strategy that can be encoded as a *(deterministic) Mealy machine*, that is, a finite-state machine that outputs a letter from O when given a letter from I. Formally, such a machine is a tuple $\mathcal{M} = (S, s_0, I, \lambda_u, \lambda_o)$ where S is a finite set of (memory) states, s_0 is the initial state, $\lambda_u \colon S \times (Q \times I) \to S$ is the update function, and $\lambda_o \colon S \times (Q \times I) \to O$ is the output function. The machine encodes σ in the following way. For all play prefixes $\langle i_1, o_1\rangle \dots \langle i_{k-1}, o_{k-1}\rangle$ and input letters $i_k \in I$, we have that $\sigma(\langle i_1, o_1\rangle \dots \langle i_{k-1}, o_{k-1}\rangle, i_k) = \lambda_o(s_k, i_k)$, where $s_{\ell+1} = \lambda_u(s_\ell, i_\ell)$ for all $\ell < k$. We then say that the strategy σ has *memory* $|S|$. In particular, when $|S| = 1$, we say that the strategy is *memoryless* (the term *positional* is also used in the literature). For all games considered in the competition, it holds that there exists a winning strategy for Eve in the game if and only if there exists a memoryless winning strategy for Eve.

2.2 Safety, parity, and linear-temporal specifications

Let I and O be finite input and output alphabets. SYNTCOMP has tracks corresponding to three different versions of the synthesis and realizability problems. The *safety tracks* correspond to these problems for games played on deterministic automata with a safety acceptance condition; the *parity tracks*, to the same problems for games played on deterministic automata with a parity acceptance condition. The remaining tracks correspond to games whose payoff set is given in the form of a *linear-temporal-logic* (LTL) formula. We explain the connection between LTL formulas and infinite-word automata in the following.

2.2.1 LTL-defined payoff sets

LTL (Pnueli [51]) is a logic that allows us to naturally specify time dependence among events that make up the formal specification of a system. Formulas in LTL are constructed from a set P of atomic propositions, the usual Boolean connectives, and temporal operators $\mathtt{X}$, $\mathtt{F}$, $\mathtt{G}$, $\mathtt{U}$, which intuitively correspond to "next", "eventually", "always", and "until" in English. Formally, LTL formulas conform to the following syntax:

$$\varphi ::= a \in P \mid \varphi \wedge \varphi \mid \neg\varphi \mid \mathtt{X}\,\varphi \mid \mathtt{F}\,\varphi \mid \mathtt{G}\,\varphi \mid \varphi\,\mathtt{U}\,\varphi$$

with derived operators such as implication defined as usual. For instance, the formula $\mathtt{G}(\mathit{req} \rightarrow \mathtt{F}\,\mathit{grant})$, over atomic propositions *req* and *grant*, can be read as "it is always the case that if there is a request, then eventually it is granted". We refer the reader to the book by Baier and Katoen [5] for the formal semantics of LTL. In the context of words over an input–output alphabet $I \times O$, the atomic propositions can be assumed to be an encoding of letters in the alphabet, that is, the truth value of the propositions is defined for each letter.

It is well known that the set $\mathrm{Words}(\varphi)$ of all words satisfying a given LTL formula φ can be "compiled" into an infinite-word automaton. For instance, we can construct (in exponential time) a nondeterministic automaton $\mathcal{N}$ with a Büchi acceptance condition such that $\mathcal{L}(\mathcal{N}) = \mathrm{Words}(\varphi)$ (Vardi and Wolper [64]). We can also construct (in doubly exponential time) a deterministic automaton $\mathcal{N}$ with a parity acceptance condition with the same property (Piterman [50], Safra [55]).

The *LTL tracks* of the competition correspond to synthesis and realizability problems that can, for example, be solved by playing games on a deterministic parity automaton compiled from a given LTL formula or by other solutions that rely on different automata constructions. In particular, algorithms for both problems exist, which avoid the expensive construction of a deterministic automaton and can work with nondeterministic automata.

2.2.2 Specification formats

We now briefly touch on the encoding used by the competition to represent the input for the safety, parity, and LTL tracks.

Safety specifications. To represent (the transition relation of) deterministic automata with a safety acceptance condition, we use And-Inverter Graphs (AIGs). In turn, to encode AIGs, we use an extended version of the AIGER format (Biere [8]). The latter is the standard format in the hardware model checking competition (Biere [7]). The main reason that the basic format has to be extended is to allow for the partitioning of the alphabet into I and O (Jacobs [36]).

Parity specifications. For automata with a parity acceptance condition, we use an extended version of the Hanoi Omega-Automata (HOA) format (Babiak et al [4]). The HOA format is a flexible exchange format for infinite-word automata. Just like the AIGER format, extending it is necessary to be able to include the partitioning of alphabet into I and O (Pérez [49]).

LTL specifications. Finally, to represent LTL specifications, we use the Temporal Logic Synthesis Format (TLSF) (Jacobs et al. [39]). TLSF allows us to define *families* of LTL specifications via parameters. Additionally, it allows us to use high-level constructs, such as sets and functions, to provide a compact and human-readable representation.

2.2.3 Output format

For the synthesis tracks, tools are expected to produce a strategy if the specification is realizable. AIGER is the format used by the competition to encode the Mealy machine implementing the strategy. In this case the standard AIGER format is sufficient.

2.3 Rules

All tracks are divided into subtracks for realizability checking and synthesis and into two execution modes, sequential (using a single core of the CPU) and parallel (using up to 4 cores). Every tool can run in up to three configurations per subtrack and execution mode. Before the competition, all tools are tested on a small benchmark set, and authors can submit bugfixes if problems are found. Tools submitted by the organizers are not allowed to submit bugfixes.

Disqualification During the competition, **no erroneous results are allowed**. For the realizability subtracks, this is easy to check. For the synthesis subtracks, we model check all synthesized strategies. In the exceptional case that the output of a tool cannot be model checked, e.g., because of it being too large for it to be analyzed within reasonable time and space, it is assumed to be correct.[3]

Extraordinary comparisons There are two cases in which an unofficial comparison run is launched. First, tools or bugfixes may be submitted after the competition is over. Second, for the purpose of including in the comparison tools that are no longer maintained by their authors, the organizers of SYNTCOMP sometimes modify or compile a (previous version) of the tool themselves. When reporting the results of a competition, we include tools that fit these two cases but refer to their results as being *hors concours*, that is, they do not officially participate in the competition.

2.3.1 Ranking schemes

In all tracks, there is a ranking based on the number of correctly solved problems within a 3600 s timeout per benchmark. In the synthesis tracks, correctness of the solution additionally has to be confirmed by a model checker. Moreover, in synthesis tracks, there is a ranking based on the quality of the solution, measured by the number of gates in the produced AIGER circuit. To this end, the size of the solution is compared to the size r of a reference solution. A circuit of the same size is rewarded 2 points, and smaller or larger solutions are awarded more or less points, respectively (see, e.g., Jacobs et al. [40] for more detail).

2.3.2 Selection of benchmarks

In 2018, benchmarks were selected according to the same scheme as in the previous years, based on a categorization into different classes (again, see Jacobs et al. [40]). From 2019 onward, all available benchmarks in the SYNTCOMP library (see Sect. 3) were used. In the LTL track, some of the TLSF-encoded benchmarks represent a family of benchmarks that can be scaled up in one or more parameters. For these, more instances are generated whenever all the existing ones are considered "too easy" for more than one tool.

2.4 Execution

In 2018, SYNTCOMP was run at Saarland University. Benchmarking was again organized on the EDACC platform (Balint et al [6]). Since 2019, the competition has been run on StarExec (Stump et al. [59]). This has had a few consequences. The main such consequence is that not all legacy tools were successfully migrated: some of them relied on deprecated packages, whereas others were implemented in languages with limited compiler support. Furthermore, such tools lacked an active maintainer.

[3] In practice, this only happened a handful of times.

Table 1 Statistics regarding the LTL benchmarks: all generated formulas are fully parenthesized. Thus by the formula length we mean the number of characters and by the formula depth the maximal nesting of temporal operators (obtained using Spot's `ltlfilt` utility). Finally, all values are presented in the format "min ≤ median, mean ≤ max"

Year	No. of bench.	Length of LTL formula	Formula depth	No. of inputs	No. of outputs	Solved by winner
2018	286	$28 \le 349, 682 \le 9358$	$1 \le 2, 3 \le 19$	$1 \le 4, 5 \le 25$	$1 \le 2, 3 \le 14$	93%
2019	346	$28 \le 464, 1031 \le 12843$	$1 \le 2, 3 \le 37$	$1 \le 4, 5 \le 25$	$1 \le 2, 4 \le 37$	94%
2020	346	$28 \le 464, 1031 \le 12843$	$1 \le 2, 3 \le 37$	$1 \le 4, 5 \le 25$	$1 \le 2, 4 \le 37$	98%
2021	942	$23 \le 896, 2031 \le 330512$	$0 \le 3, 4 \le 37$	$1 \le 4, 7 \le 70$	$1 \le 3, 5 \le 64$	90%

Our use of StarExec has significantly simplified the organizational effort, while admittedly raising the entry threshold for participants a bit, since they can, and must, themselves make sure that their code runs on the competition servers. In practice, all tools that were still actively maintained (more precisely, tools for which a PhD student worked on the tool) were updated and migrated to StarExec. Any reduction in the number of participating tools from 2018 to 2019 is instead due to a number of projects ending and PhD students graduating around the time. For an up-to-date overview of the specifications of the StarExec service, we refer the reader to its website[4] and wiki.[5]

3 Benchmarks

Since the inception of SYNTCOMP, benchmarks in different specification formats have been collected in the SYNTCOMP library.[6] The benchmarks in the SYNTCOMP library mainly come from three sources: i) participants are invited to submit benchmarks from their own research, ii) benchmarks from the literature are translated into our standard format, and iii) we use different techniques to automatically generate additional benchmarks.

In the following, we shortly comment on families of benchmarks that were added to each track during the relevant years.

3.1 LTL

Between 2018 and 2019, several *temporal-stream logic* (TSL) benchmarks were submitted to the LTL tracks of SYNTCOMP. TSL is a new temporal logic that allows us to separate control and data (Finkbeiner et al. [32]). Among others, the logic can be used to specify components of games implemented for FPGAs (Geier et al. [35]) and to specify functional reactive programs (Finkbeiner et al. [31]). Importantly, bounded TSL specifications can be translated into LTL specifications—this is how the benchmarks are obtained.

In addition to the TSL benchmarks, Felix Klein also submitted LTL benchmarks based on hardware-component specifications and an encoding of an "infinite-duration tic-tac-toe". In 2020 and 2021, TLSF files encoding families of LTL specifications were used to generate more challenging benchmarks, i.e., larger parameter values were used to generate more difficult instances of each family to gauge how well tools scale with respect to the parameters.

In Table 1, we summarize some statistics of interest regarding the benchmarks used for all relevant editions of the LTL tracks. All data used to generate the table were fetched from https://syntcomp.react.uni-saarland.de/ and https://github.com/SYNTCOMP/benchmarks/releases.

3.2 Safety

In 2019, random benchmarks in the extended AIGER format were systematically generated by Mouhammad Sakr by uniformly sampling from the set of all specifications with given values for the number of (un)controllable inputs and latches. The techniques used for this were subsequently improved and described by Jacobs and Sakr [38].

3.3 Parity games

In 2020, Spot (Duret-Lutz et al. [24]) was used to generate deterministic parity automata from some LTL benchmarks. It is important to mention that, perhaps because this translation is doubly exponential in general, not many specifications could be translated. Furthermore, the ones that did yield a deterministic parity automaton resulted in small automata. To be precise, 121 benchmarks were used for that first edition of the parity-game track, and they were all generated as previously mentioned.

In 2021, more benchmarks were translated as described above (for a total of 303 benchmarks generated in that way). Also, 217 combinatorially hard benchmarks were generated using the PGSolver parity-game suite (Friedmann and Lange [34]) and then translated to the extended HOA format.

[4] https://www.starexec.org/.

[5] https://wiki.uiowa.edu/display/stardev/User+Guide.

[6] https://github.com/SYNTCOMP/benchmarks.

Table 2 Participation years, authors, and links to source code for tools mentioned in this paper

Tool & participation years	Developers	URL	Section
Acacia bonsai	Cadilhac et al.	https://github.com/gaperez64/acacia-bonsai	Sect. 4.1
BoSy ('17, '18)	Faymonville et al.	https://www.react.uni-saarland.de/tools/bosy/	Sect. 4.2
BoWSer ('17, '18)	Finkbeiner et al.	https://www.react.uni-saarland.de/tools/bowser/	Sect. 4.3
Knor ('20–'21)	Tom van Dijk	https://github.com/trolando/knor	Sect. 4.4
LazySynt	Sakr et al.	https://github.com/mhdsakr/Lazy-Safety-Synthesis	Sect. 4.5
Ltlsynt ('17–'21)	Renkin et al.	https://spot.lrde.epita.fr/ltlsynt.html	Sect. 4.6
`Otus` (2021)	Abraham et al.	https://doi.org/10.5281/zenodo.5046346	Sect. 4.7
Party/Kid & `sdf` ('17, '18, '21)	Ayrat Khalimov	https://github.com/5nizza/sdf-hoa	Sect. 4.8
Simple BDD Solver ('14–'21)	Adam Walker	https://github.com/adamwalker/syntcomp	Sect. 4.9
SPORE (2021)	Bruyère et al.	https://github.com/Skar0/spore	Sect. 4.10
`Strix` ('18–'21)	Meyer et al.	https://github.com/meyerphi/strix	Sect. 4.11

4 Updated participating tools

In this section, we give an overview of the participants of the 2018–2021 editions of SYNTCOMP. All of them can be found in Table 2.

In the following, we mostly focus on the participants of the parity and LTL tracks. The safety track had minimal active participation: in 2018, the only tool that received an update was Simple BDD Solver, and the only new tool was LazySynt, which participated *hors concours*. No updates or new tools were submitted to the safety track after 2018.

Some of the tool descriptions below may refer to specialized techniques. We refer the interested reader to the publications cited in the text for further details on such techniques. After the overview of the participants, we conclude this section with a classification of the LTL-synthesis tools based on techniques, data structures, and algorithms.

4.1 Acacia bonsai

Acacia bonsai, the spiritual successor of Acacia+ (Bohy et al. [11]), participated hors concours in the 2021 edition of the LTL-realizability track. It implements *downset*-based algorithms (i.e. algorithms that manipulate downward closed sets) that avoid constructing a deterministic automaton for the given LTL specification. Instead, the downsets are used to efficiently store sets of states in an on-the-fly determinization process. These algorithms were introduced by Filiot et al. in the 2010s and implemented in the tools Acacia and Acacia+ in C and Python (Bohy [10]). Acacia bonsai is a complete rewrite of Acacia in C++20, articulated around *genericity* (that is, a library of downset functions with generic type parameters) and leveraging modern techniques for better performance. These techniques include compile-time specialization of the algorithms, the use of SIMD registers to store vectors, and several preprocessing steps, some relying on efficient Binary Decision Diagram (BDD) libraries. It also includes different data structures to store downsets such as *k-d trees*, a useful data structure for organizing points in a k-dimensional space (see, e.g., de Berg et al. [23]).

It is worth mentioning that to compile the input LTL formula into an automaton, Acacia bonsai uses Spot (Duret-Lutz et al. [24]).

4.2 BoSy

BoSy was updated by P. Faymonville, B. Finkbeiner, and L. Tentrup in 2018 and competed in both the realizability and the synthesis track. To detect realizability, BoSy translates the (complement of the) LTL specification into a safety automaton by bounding the number of visits to Büchi states. The resulting safety game is solved by SafetySynth. For synthesis, BoSy relies on an encoding into quantified Boolean formulas (QBF). A full account of the algorithms implemented in the tool is given by Faymonville et al. [29]. Two configurations of BoSy competed in SYNTCOMP 2018, configuration (basic) and configuration (opt), where the latter further improves the size of the strategy by encoding the existence of an AIGER circuit representing the strategy directly into a QBF query. Both configurations support a parallel mode if more than one core is available.

4.3 BoWSer

BoWSer was updated by B. Finkbeiner and F. Klein in 2018. It implements different extensions of the *bounded synthesis* approach that solves the LTL synthesis problem by first translating the complement of the specification into a Büchi automaton and then encoding acceptance of a transition system with bounded number of states into a constraint system, in this case a propositional satisfiability (SAT) problem. The details of all encodings are described by Finkbeiner and Klein [30].

Compared to 2017, a number of small improvements to speed up computations were implemented, and an experimental preprocessor to simplify LTL formulas has been added. The sequential configurations of the tool spawn multiple threads that are executed on a single CPU core. The parallel configurations are mostly the same as the sequential ones but use a slightly different strategy for exploring the search space of solutions.

4.4 Knor

Knor is a BDD-based solver for parity specifications first submitted by T. van Dijk in 2020. It leverages the Sylvan BDD package (van Dijk and van de Pol [63]) and the Oink parity-game solver (van Dijk [61]).

Knor implements a translation from HOA to a symbolic parity automaton encoded using BDDs. Importantly, the chosen variable ordering encodes first source states, then uncontrollable inputs, controllable inputs, and finally the target states. The resulting symbolic parity automaton can then be treated in two ways. First, it can be solved directly using a symbolic parity game algorithm (Lijzenga and van Dijk [43]) optimized to utilize the aforementioned variable ordering. This solution immediately yields a controller that can be dumped as an AIG. Alternatively, the symbolic parity automaton can be output as an explicit parity game. (Note that the previous encoding into BDDs might have reduced the size of the automaton, exponentially in some cases.) Such an explicit parity game can then be solved by any of the algorithms implemented in Oink. This last option does not yet support synthesis, only realizability.

For the first solution, synthesis is realized using a naïve construction of an AIG realizing the Mealy controller, based on a simple application of the Shannon expansion of the BDD-encoded functions.

4.5 LazySynt

The *Symbolic Lazy Synthesis* (LazySynt) tool was submitted in 2018 by M. Sakr and S. Jacobs. It participated *hors concours* in the safety-synthesis track. In contrast to the classical BDD-based algorithm and the SAT-based methods implemented in Demiurge (Bloem et al. [9], Seidl and Könighofer [56]), LazySynt implements a combined forward–backward search embedded into a refinement loop, generating candidate solutions that are checked and refined with a combination of backward model checking and forward generation of additional constraints (Jacobs and Sakr [38]).

4.6 Ltlsynt

The program `ltlsynt`, introduced to SYNTCOMP in 2017 (see Table 3), is part of Spot (Duret-Lutz et al. [24]). It relies on a translation of the LTL specification to a parity game whose winning strategy is then encoded as an AIGER circuit. The version submitted to the 2021 edition features the following improvements (Renkin et al. [53]):

- A decomposition of the input specification when possible (Finkbeiner et al. [33]).
- An LTL translation to Deterministic Emerson–Lei Automata (DELA) that handles various simplifications: splitting the input formula in a manner similar to the `delag` tool (Müller and Sickert [48]), detecting *obligation* subformulas (Esparza et al. [27]), relying on weak automata and suspendable properties (Babiak et al [3]).
- An SCC-based paritization algorithm (Renkin et al. [52]) for DELA that relies on a *color appearance record*.
- A transition-based parity game solver adapted from van Dijk [61], supporting (nonrecursive) SCC decomposition and parity compression.
- Optimization of the winning strategy through a variant of Spot's simulation-based reduction based on BDD signatures or an improvement of a SAT-based minimization algorithm for Incompletely Specified Mealy Machines (Renkin et al. [54]).

4.7 Otus

`Otus` (Abraham [1]) is a tool for LTL synthesis using symbolically represented parity automata and games. It proceeds as follows: the LTL formula is decomposed into a Boolean combination of *simpler* formulas, and these formulas are separately translated to BDD-encoded deterministic automata and then recomposed by computing the (deterministic) union and intersection on the BDD-representation. Concretely, `Otus`

1. makes use of the Δ_2-normalization and the translation to deterministic (co-)Büchi automata found in (Sickert and Esparza [57]) and implemented by `Owl` (Kretínský et al. [42]),
2. computes the symbolic representation of a deterministic Rabin automaton by union and intersection, and
3. applies a symbolic implementation (Boker et al. [12]) to obtain a parity automaton.
4. This symbolic automaton is reinterpreted as a parity game, which is then solved by a symbolic implementation of the distraction fix-point iteration (Lijzenga and van Dijk [43], van Dijk and Rubbens [62]).

To speed up the BDD-operations, `Otus` makes use of the Sylvan BDD package (van Dijk and van de Pol [63]).

4.8 Party/Kid and `sdf`

Party/Kid and `sdf`, submitted by A. Khalimov, implement variants of symbolic bounded synthesis (Ehlers [25, 26]).

Table 3 Versions of Spot on which `ltlsynt` submissions to SYNTCOMP were based

Year	Version	Main changes in `ltlsynt`
2017	pre-2.4	first implementation
2018	2.5.3	optimizations to determinization and game solving; incremental determinization approach
2019	2.7.4	(bugged) latest appearance record (LAR); improved LTL translation; incremental determinization removed
2020	2.9	reimplemented LAR, split, and game solving; parity minimization
2021	2.9.7	input decomposition; strategy simplification; specialized strategy construction for some LTL fragments

Both tools run two tasks, realizability and unrealizability, in parallel. We further describe how the realizability check is done. Unrealizability can be checked in a similar fashion by adequately modifying the given LTL formula.

First, the tool translates the given LTL formula into a universal co-Büchi automaton (UCW) using the TLSF-manipulation tool `syfco` (Jacobs et al. [39]) and the Spot automata library (Duret-Lutz et al. [24]). Then it iterates over increasing bounds on the number of visits to final states of the UCW: given such a bound, it translates the UCW into a universal safety automaton. The universal safety automaton is then encoded into a safety game in a BDD-based representation (using the CUDD library (Somenzi [58])), where each state of the universal automaton gets a separate variable in the BDDs, thus avoiding explicit determinization. The game is then solved using the standard fix-point algorithm. The strategy extraction is also standard and does not use any third-party tools.

Over the years 2017, 2018, and 2021, three versions participated, with only technical differences. The latest version (`sdf`, 2021) is written in C++.

4.9 Simple BDD solver

An update of Simple BDD Solver, submitted by Adam Walker, competed in the 2018 safety-realizability track. Simple BDD Solver implements the classical BDD-based fix-point algorithm for safety games (Jacobs et al. [41]). In sequential mode, it runs in three configurations, two of which are based on an abstraction-refinement approach inspired by de Alfaro and Roy [22] and one without any abstraction. All three implement many important optimizations. These configurations are the same as in 2017. Additionally, three new configurations were entered for the parallel mode. These run different portfolios of the algorithms in the sequential mode.

4.10 SPORE

SPORE is a prototype tool designed to assess the viability of using *generalized parity games* for LTL realizability. These are games played on automata with multiple priority functions requiring all of the corresponding parity conditions to be satisfied. The input LTL formula is first decomposed into a conjunction of subformulas, which are in turn translated into deterministic parity automata, composed into a synchronized (generalized parity) product automaton, and finally translated into a generalized parity game using the `tlsf2gpg`[7] tool. The game is then solved using a combination of the recursive algorithm for generalized parity games (Chatterjee et al. [18]) and incomplete polynomial-time algorithms, called partial solvers, presented by Bruyère et al. [13]. SPORE contains an explicit and a symbolic implementation of those algorithms, the latter relying on BDDs to represent the game arena. The version presented in 2021 is implemented in Python with dd[8] as a library to manipulate BDDs.

4.11 Strix

`Strix` is a tool for LTL synthesis using transition-based deterministic parity automata (tDPW) and parity games as intermediate steps. Since its inception (Meyer et al. [47]), it proceeds in four stages: 1) formula rewriting and decomposition, 2) automaton construction, 3) winning strategy computation, and 4) controller extraction. Note that stages 2 and 3 run in parallel, are on-the-fly, and exchange information, so that the automaton construction can be focused on critical states and early termination is possible.

The improvements applied for SYNTCOMP 2019 and 2020 are described by Luttenberger et al. [45] and include a refined controller extraction stage and updates to the LTL-translations based upon the work of Esparza et al. [28] and provided by `Owl` (Kretínský et al. [42]). For SYNTCOMP 2021, the following major changes have been applied (Meyer and Sickert [46]):

- LTL formulas are now translated to transition-based deterministic Emerson–Lei automata (tDELA) by combining constructions (Δ_2-normalization, direct translation to deterministic automata) from Sickert and Esparza [57] with a product construction adapted from Müller and Sickert [48]. Then either a tDELA-to-tDPW construction based

[7] See https://github.com/gaperez64/tlsf2gpg.

[8] See https://github.com/tulip-control/dd.

on Zielonka-trees or the Alternating Cycle Decomposition is applied (Casares et al. [16, 17]).
- The strategy iteration algorithm, by Luttenberger [44], for solving parity games has been replaced by a Rust implementation of the distraction fixpoint iteration algorithm (DFI) (van Dijk and Rubbens [62]).

4.12 Classification of LTL tools

To summarize some of the tool descriptions given above, the participants of the competition can be classified as implementing one of two approaches.

Bounded synthesis. They translate the LTL specification into a universal co-Büchi automaton. Then, for an increasingly larger bound $b \in \mathbb{N}$, either (i) the automaton is turned into a universal safety automaton by allowing at most b visits to rejecting states and then determinized, or (ii) the search is limited to winning strategies encoded as a Mealy machine with at most b memory states. The resulting problems are then solved either by standard techniques for safety games or by encoding them into a constraint system and employing a SAT-, QBF-, or SMT-solver.

The Owl–Pig approach. We coin the class name *Owl–Pig* to refer to solvers that first translate the LTL specification into a deterministic parity automaton using one of several algorithms described in the works of S. Sickert et al. and implemented in the `Owl` tool (see Sect. 4.11). Then they solve the resulting parity game using one of several algorithms described in the works of T. van Dijk et al. and implemented in the tools Oink and Knor (the sound a pig makes in English and Dutch, respectively).

Tools that implement the bounded synthesis approach mostly differ in choice of programming language, (symbolic) data structure or logic encoding, and techniques for *counter reduction*, that is, the detection of rejecting states for which knowing whether they have been visited or not is sufficient (i.e., instead of storing the exact number of visits).

For the Owl–Pig approach, generating in full the deterministic parity automaton is clearly the bottleneck. Hence optimizations, types of automata used as intermediate steps, symbolic encodings, and on-the-fly constructions are the main differences among those tools. We present our classification and summarize some noteworthy differences between them in Table 4 and Table 5. We concede that our classification is not perfect: for instance, SPORE does not really implement the Owl–Pig approach, yet we classify it as such since it does go through a similar two-stage algorithm: translate the specification to a deterministic automaton and solve a game.

5 Rankings and experiments

In this section, we elaborate on the results of each track of the relevant editions of SYNTCOMP. All data used for the graphs and analyses given below were fetched from https://syntcomp.react.uni-saarland.de/ for the 2018 edition and https://www.starexec.org/starexec/secure/explore/spaces.jsp?id=329383 for later editions. We have also archived a copy of all scripts and data used by Jacobs and Perez [37].

Table 4 Differences between bounded synthesis tools

Tool	Symbolic data structures/encodings
Acacia bonsai	Antichains
BoSy	BDDs, SAT, QBF
BoWSer	SAT
Party/Kid & `sdf`	BDDs

5.1 Safety track

Despite the update submitted for the Simple BDD Solver tool, the tool rankings did not change in 2018 compared to 2017. We refer the interested reader to the SYNTCOMP'17 report (Jacobs et al. [40]). In the following years, the safety track has seen neither new participants nor updates to the existing tools.

5.2 Parity track

In 2020 and 2021, two tools participated in the parity tracks, `Strix` and Knor. The former is, oversimplifying, the parity-game solving component of the tool `Strix` that is currently dominating the LTL track. Knor implements several classical and novel parity-game solving algorithms and combinations thereof. In 2020, both tools competed in the realizability track only. In 2021, they competed in the synthesis track only and in a special *combinatorially hard* realizability track. For these two initial editions of the parity tracks, no distinction was made between sequential and parallel subtracks.

The results are summarized in Table 6 and Table 7. Probably, the most important insight from the results of these tracks is that Knor-BDD—which only participated *hors concours* in 2020 (since it was submitted after the official deadline) but later became the default configuration of Knor—outperformed both `Strix` and Knor by almost an order of magnitude only by switching to a symbolic representation of the input game. Here participants noted that most benchmarks can be solved easily and that the current bottleneck is parsing the input and constructing an internal representation of the game. This explains the initial advantage Knor-BDD exhibited in 2020.

An additional point of interest is that Knor-BDD solved *all* benchmarks in 2020, 276 out of 303 in 2021, and 216 combinatorially hard benchmarks out of a total of 217 in the same year.

Table 5 Differences between Owl–Pig tools

Tool	Automata used	Game-solving algorithms
Ltlsynt	Det. Emerson–Lei, Parity	Transition-based parity-game solver (van Dijk [61])
`Otus`	BDD-encoded Büchi, Rabin, Parity	Symbolic DFI (Lijzenga and van Dijk [43], van Dijk and Rubbens [62])
SPORE	Generalized Parity	Recursive + partial solvers (Bruyère et al. [13], Chatterjee et al. [18])
`Strix`	Det. Emerson–Lei, Parity	Distraction fixpoint (DFI) (van Dijk and Rubbens [62])

Table 6 Results of Parity Realizability Track 2020–2021

Tool	2020		2021
	Solved/total	Solving time (s)	Solved/total
Strix	**122/122**	**6.84**	150/217
Knor	**122/122**	12.71	-
Knor-BDD	(122/122)	(1.57)	**216/217**

Table 7 Results of Parity Synthesis Track 2021

Tool	Solved/total	Score
Strix	260/303	**374.89**
Knor-BDD	**276/303**	252.66

5.3 LTL track

The LTL track has been running since 2016, and the set of actively participating (i.e., new and updated) tools has changed a bit since. In Fig. 1, we depict the changes in the LTL-realizability rankings every year since 2017, and in Table 8, we give an overview of the results in both the realizability and synthesis subtracks from 2018 to 2021.

Here, and in similar graphs shown in the rest of this paper, we have selected the best configuration submitted per tool. Additionally, all tools—regardless of whether they were parallel or sequential—were assigned as score the number of benchmarks solved within the same time limits. It is worth noting that the best tool in 2017 implemented bounded synthesis algorithms, whereas in recent years the competition has been dominated by parity-game-based tools, which focus on optimizing the LTL-to-automaton compilation. Namely, `Strix` has remained first in all rankings since 2018.

Regarding 2021, in Fig. 2, we have plotted the total amount of time it takes for each tool to solve increasing numbers of benchmarks. Once more, we do not distinguish between parallel and sequential tools. For reference, we have also included Acacia bonsai in the plot, even if it only participated *hors concours*.

In what follows, we analyze the time and quality rankings of the tools in the synthesis subtrack of the 2018 and 2021 editions of SYNTCOMP. Note that from 2019 to 2020 only ltlsynt and `Strix` participated, and they did so too in 2021, hence our choice of representative years. Additionally, we present the state of the art in terms of scalability for different parameterized families of benchmarks we have used for the competition.

5.3.1 Synthesis subtrack 2018, 2021

In Fig. 3, we have plotted, for each tool, the total amount of time it takes for it to solve increasing numbers of benchmarks. Here we included configurations from the previous years of tools, which were updated and Party (as legacy tool) for reference. Additionally, both realizable and unrealizable benchmarks were counted for the score. The previous results should be compared with Fig. 4, where we have plotted the total size of the outputs generated for increasing numbers of (realizable) benchmarks.

5.3.2 Parameterized families of benchmarks

Presently, we focus on families of LTL specifications defined by TLSF benchmarks with parameters. In 2021, LTL specifications coming from 26 such TLSF files were used in the competition. For most of those families, the results are consistent with the previously summarized results, i.e., the rankings of the tools are preserved when restricting the score to solved benchmarks in that set only. However, some exceptions do exist.

For instance, the families of combinatorial logic specifications `mux` and `shift`, which specify transducers realizing a mutiplexer and a barrel shifter, respectively, are solved most efficiently by tools other than `Strix`. See Fig. 5 for the relevant plots. Other exceptions include the `collector_v2` and `detector` families of benchmarks; see Fig. 6. For the former, the (total) solving time for the best tool is more than order of magnitude smaller than that of `Strix`. Other interesting families include arbiter specifications such as the `round_robin_arbiter` and the `full_arbiter` families. There `Strix` is still best overall, but not consistently so; see Fig. 7.

5.3.3 Solved benchmark statistics

Finally, we also give some statistics of the benchmarks solved in the 2021 edition of the competition. We present statistics for the benchmarks solved by all participating tools,

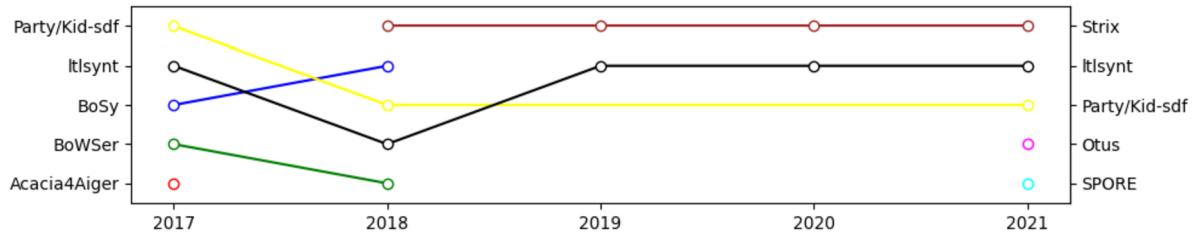

Fig. 1 Bump plot of the rankings for the LTL realizability tracks for all editions of SYNTCOMP 2017–2021 (2017 is included for reference since some tools are no longer maintained and being updated)

Table 8 Results of LTL Realizability & Synthesis Track 2018–2021

Tool	2018		2019		2020		2021	
	Solved (out of 286)	Score	Solved (out of 434)	Score	Solved (out of 434)	Score	Solved (out of 924)	Score
`Strix`	**267**	**446**	**418**	**717**	**424**	**600**	**827**	**793**
BoSy	244	402	-	-	-	-	-	-
Party/Kid & `sdf`	242	-	-	-	-	-	730	448
`ltlsynt`	239	258	361	349	398	360	745	543
BoWSer	212	315	-	-	-	-	-	-
Otus	-	-	-	-	-	-	542	249
Spore	-	-	-	-	-	-	499	-

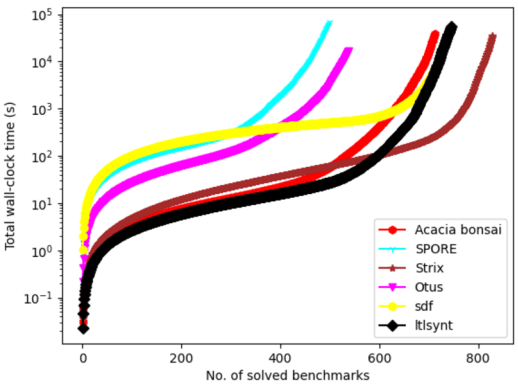

Fig. 2 Cactus (a.k.a. survival) plots for the participants of the LTL realizability track of SYNTCOMP 2021; Note that the y-axis is displayed using the logarithmic scale

those solved by tools implementing the bounded synthesis approach, and those solved by tools implementing the Owl–Pig approach. The data are given in Table 9. This should be compared with the data presented in the last row of Table 1.

We observe that the set of benchmarks bounded synthesis tools are able to solve is longer. Further, benchmarks not solved by bounded synthesis tools have deeper average temporal-operator nesting than those not solved by Owl–Pig solvers. This is in line with the intuition that complicated LTL formulas may yield complex and larger deterministic automata, thus slowing down Owl–Pig solvers whose bottleneck is exactly constructing the automaton. In contrast, recall that bounded synthesis tools avoid directly constructing a deterministic automaton. Instead, bounded synthesis tools are mostly affected by how large the bound b needs to grow to find a solution (i.e., how long the satisfaction of liveness properties has to be postponed, or how many states the smallest possible solution has). For instance, the developers of Acacia bonsai report (Cadilhac and Pérez [14]) on the bound required for their tool to solve 667 of the benchmarks[9] from the 2021 edition of SYNTCOMP: From those, 546/667 finish with $b = 2$, and 106 more with $2 < b \leq 5$, so that only 15 (of the solved benchmarks) need a bound larger than 5 and none required more than $b = 8$.

Interestingly, based on the statistics of benchmarks not solved by Owl–Pig solvers, we could conclude that the number of inputs also affects these tools more than it does bounded synthesis ones.

6 Conclusion

In this paper, we have reported on the last four editions of SYNTCOMP, the reactive synthesis competition. Further-

[9] To be precise, their experiment also fixes a timeout of around one minute. This means the numbers that follow correspond to a subset of benchmarks that Acacia bonsai can solve very quickly.

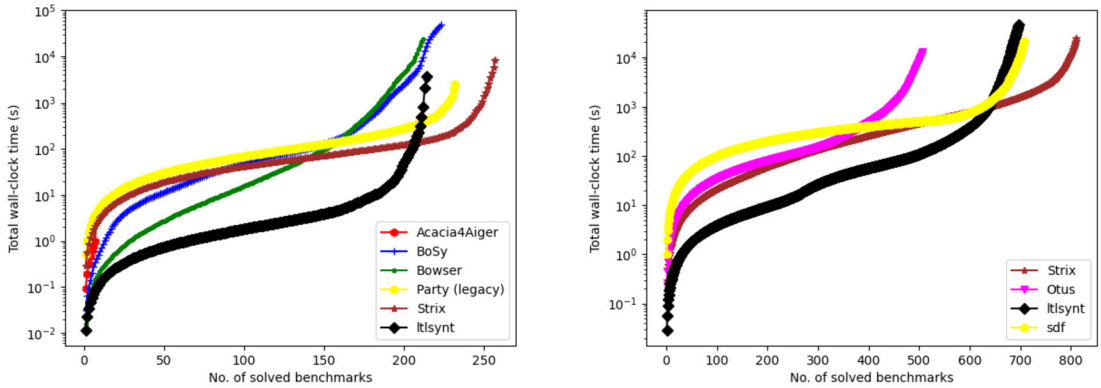

Fig. 3 Cactus (a.k.a. survival) plots for the participants of the LTL synthesis track of SYNTCOMP 2018 (left) and 2021 (right); Again, the y-axis is displayed using the logarithmic scale

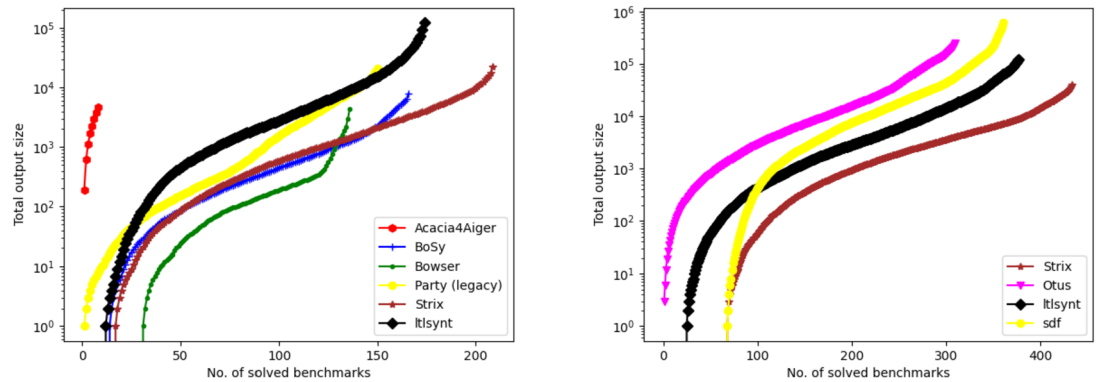

Fig. 4 Cactus plots for the participants of the LTL synthesis track of SYNTCOMP 2018 (left) and 2021 (right)—this time, showing total output size instead of time (counting AND-gates only); Note that the y-axis is displayed using the logarithmic scale

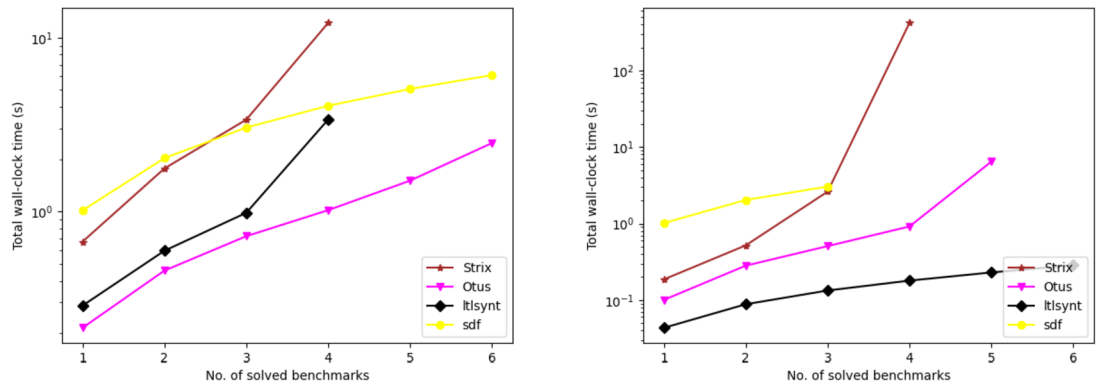

Fig. 5 Cactus plots for all the participants of the LTL synthesis track of SYNTCOMP 2021, restricted to the `mux` (left) and `shift` (right) benchmark families

more, we analyzed the results of our experimental evaluations, including a ranking of the participating tools with respect to quantity and quality of solutions. We observe a measurable improvement in terms of the performance of the top solvers in the LTL and parity tracks (see Table 8 and Table 9, then Table 6 and Table 7, respectively). In particular, every year we see more tools solving more benchmarks. Additionally, the number of collected benchmarks continues to grow (see Table 1 and Table 6). It is also important to note that the SYNTCOMP benchmarks are now being used in subfields of computer science outside of verification: In the field of satisfiability testing, synthesis can be seen as an application

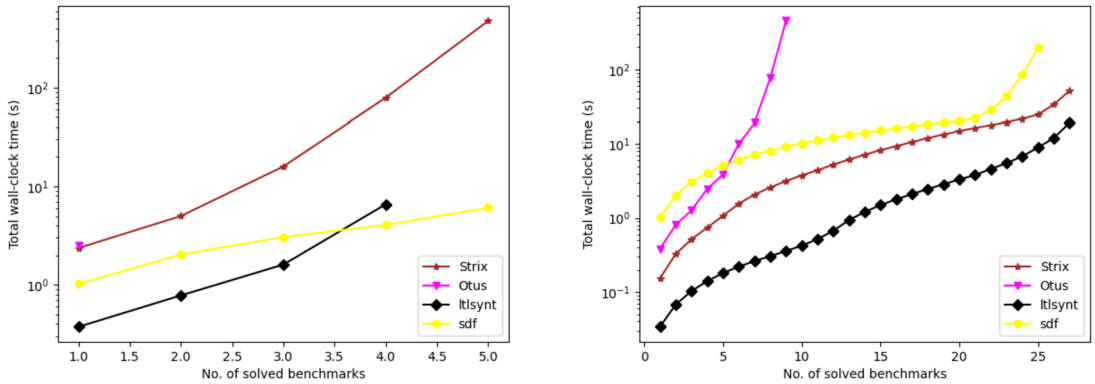

Fig. 6 Cactus plots for all the participants of the LTL synthesis track of SYNTCOMP 2021, restricted to the `collector_v2` (left) and `detector` (right) benchmark families

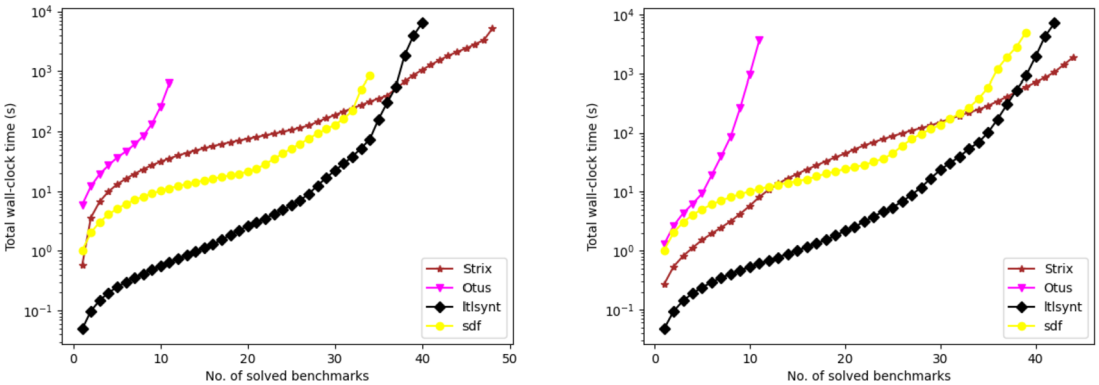

Fig. 7 Cactus plots for all the participants of the LTL synthesis track of SYNTCOMP 2021, restricted to the `round_robin_arbiter` (left) and `full_arbiter` (right) benchmark families

Table 9 Statistics regarding all LTL benchmarks (not) solved by each solver class in 2021. The first column indicates the subset of benchmarks under consideration. Other notational conventions are the same as in Table 1. The last row is reproduced from Table 1 for comparison

	No. of bench.	Length of LTL formula	Formula depth	No. of inputs	No. of outputs
Solved by ≥ 1 tool	869	$23 \leq 804, 1538 \leq 74236$	$0 \leq 2, 4 \leq 37$	$1 \leq 4, 6 \leq 70$	$1 \leq 3, 5 \leq 64$
Solved with b. synth.	761	$23 \leq 720, 1499 \leq 74236$	$0 \leq 2, 4 \leq 22$	$1 \leq 4, 6 \leq 70$	$1 \leq 3, 4 \leq 64$
Solved with Owl–Pig	842	$23 \leq 786, 1276 \leq 17416$	$0 \leq 2, 4 \leq 37$	$1 \leq 4, 6 \leq 70$	$1 \leq 3, 5 \leq 64$
Not with b. synth.	181	$116 \leq 1403, 4271 \leq 330512$	$2 \leq 4, 7 \leq 37$	$1 \leq 6, 9 \leq 65$	$1 \leq 5, 7 \leq 64$
Not with Owl–Pig	100	$261 \leq 3282, 8392 \leq 330512$	$2 \leq 3, 6 \leq 17$	$3 \leq 10, 12 \leq 37$	$1 \leq 6, 8 \leq 30$
All benchs.	942	$23 \leq 896, 2031 \leq 330512$	$0 \leq 3, 4 \leq 37$	$1 \leq 4, 7 \leq 70$	$1 \leq 3, 5 \leq 64$

for *quantified Boolean formula* tools (see, e.g., Tentrup and Rabe [60], where the SYNTCOMP benchmarks are used as a reference). In the field of artificial intelligence, synthesis can be seen as an application for *planning* tools (see, e.g., Camacho et al. [15]). Finally, in the field of programming languages, reactive synthesis tools have been found useful to generate functional programs (Finkbeiner et al. [31]).

6.1 Lessons learned in LTL synthesis

In Sect. 4.12, we describe a classification of all LTL-synthesis tools into one of two classes. In 2021, `Strix` and Ltlsynt, both in the Owl–Pig class, dominate the LTL tracks. We present below a list of "successful tricks", which, we believe, have led to the current status of the ranking of the tools.

1. LTL specification rewriting and decomposition (see, e.g., Sickert and Esparza [57] and Finkbeiner et al. [33] or the descriptions of `Strix` and `ltlsynt` in Sect. 4).
2. On-the-fly/incremental or iterative automaton construction and game solving on partial automata (i.e., as implemented in `Strix`)
3. For bounded synthesis constructing a universal safety automaton: *counter reduction*, to reduce the number of states for which counting up to the bound b cannot be reduced to a Boolean flag.
4. Adaptation of state-of-the-art practical game-solving algorithms (e.g., Lijzenga and van Dijk [43], van Dijk and Rubbens [62])
5. Symbolic encodings: (parallel) BDDs, antichains, SAT or QBF

In particular, we observe that the first and second points are (in our opinion) what allowed `Strix` to take LTL-synthesis to a new level in 2018. From then onward, they have stayed ahead of other tools by focusing on those same points together with the fourth one. On the other hand, bounded synthesis with incremental automata constructions or counter reduction based on rewriting or decomposing the LTL specification has not received as much attention, though it was mentioned as an interesting avenue in earlier works (Ehlers [26]). However, since for bounded synthesis, the construction of the automata is not the bottleneck, these techniques may not be as valuable. Additionally, we believe that techniques for finding the right stratification of the search space by bounding certain parameters (not only memory used or visits to rejecting states) offer other avenues of potential improvements and have only been explored to a small extent.

Finally, successful tricks seem to be easy to transfer between tools of the same class. As an example, we note that from 2017 to 2018, BoSy integrated the approach that allowed Party/Kid to win in 2017. This allowed BoSy to overtake Party/Kid in 2018. Admittedly, such transfers might be harder between bounded synthesis tools that use different symbolic data structures or encodings.

6.2 Lessons learned in parity games

From the results of the 2020 and 2021 editions of the competition it is clear that the parity-game tracks require more interesting benchmarks to attract more participants and to make the competition more interesting. Currently, all tools implement similar parity-game solving algorithms, and the only advantage Knor seems to have is in terms of the representation of the game and the use of its own (parallel) BDD engine.

For parity games derived from practical examples, purely symbolic algorithms perform very well; for contrived artificial constructions, solving an explicit parity game (e.g., in Oink) is superior, though the choice of explicit algorithm makes a big difference.

6.3 The future of SYNTCOMP

In future editions, we will try to replicate the success of the new parity-game track. Indeed, several teams submitting solvers to it eventually ended up extending their tool to participate in the LTL tracks as well. Finally, we will also update some of the rules of the competition so that it remains interesting to the community.

Changes to rules Starting with the 2022 edition of the competition, the following changes will be implemented:

Benchmark copyright. All submitted benchmarks will be made publicly available via our repository[10] under a CC-BY license.

Parallel tracks. To have clearer conclusions regarding which tools are faster, we will no longer differentiate between tool submissions to sequential and parallel tracks. Instead, we will make this distinction by having two rankings: one based on wallclock time (thus favoring parallel configurations) and one based on user-CPU time (favoring sequential ones). Regarding concrete time limits, all tools will be allowed T units of wall-clock time and $4T$ units of user-CPU time (to limit parallel configurations). The factor 4 was chosen since each *job* ran on StarExec has access to one 4-core CPU.

StarExec. For the next 3–5 years, at least, we will continue using StarExec to host the competition. We will follow its evolution in terms of hardware and will later revisit the question of whether alternative (locally hosted) frameworks better fit the needs of the competition and the community.

References

1. Abraham, R.: Symbolic LTL reactive synthesis (2021). http://essay.utwente.nl/87386/
2. Apt, K.R., Grädel, E. (eds.): Lectures in Game Theory for Computer Scientists Cambridge University Press, Cambridge (2011). http://www.cambridge.org/gb/knowledge/isbn/item5760379
3. Babiak, T., Badie, T., Duret-Lutz, A., et al.: Compositional approach to suspension and other improvements to LTL translation. In: Proceedings of the 20th International SPIN Symposium on Model Checking of Software (SPIN'13). Lecture Notes in Computer Science, vol. 7976, pp. 81–98. Springer, Berlin (2013). https://doi.org/10.1007/978-3-642-39176-7_6
4. Babiak, T., Blahoudek, F., Duret-Lutz, A., et al.: The Hanoi omega-automata format. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification – 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015. Proceedings, Part I, Lecture Notes in Computer Science, vol. 9206, pp. 479–486. Springer, Berlin (2015). https://doi.org/10.1007/978-3-319-21690-4_31
5. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press, Cambridge (2008)

[10] Currently hosted here at https://github.com/SYNTCOMP/benchmarks.

6. Balint, A., Diepold, D., Gall, D., et al.: EDACC – an advanced platform for the experiment design, administration and analysis of empirical algorithms. In: Coello, C.A.C. (ed.) Selected Papers, Learning and Intelligent Optimization – 5th International Conference, LION 5, Rome, Italy, January 17–21, 2011. Lecture Notes in Computer Science, vol. 6683, pp. 586–599. Springer, Berlin (2011). https://doi.org/10.1007/978-3-642-25566-3_46
7. Biere, A.: Hardware model checking competition (2007). http://fmv.jku.at/hwmcc/
8. Biere, A.: Aiger format and toolbox (2011). http://fmv.jku.at/aiger/
9. Bloem, R., Könighofer, R., Seidl, M.: Sat-based synthesis methods for safety specs. In: McMillan, K.L., Rival, X. (eds.) Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19–21, 2014. Proceedings, Lecture Notes in Computer Science, vol. 8318, pp. 1–20. Springer, Berlin (2014). https://doi.org/10.1007/978-3-642-54013-4_1
10. Bohy, A.: Antichain based algorithms for the synthesis of reactive systems. PhD thesis, University of Mons (2014)
11. Bohy, A., Bruyère, V., Filiot, E., et al.: Acacia+, a tool for LTL synthesis. In: Madhusudan, P., Seshia, S.A. (eds.) CAV, LNCS, vol. 7358, pp. 652–657. Springer, Berlin (2012). https://doi.org/10.1007/978-3-642-31424-7_45
12. Boker, U., Kupferman, O., Steinitz, A.: Parityizing rabin and streett. In: Lodaya, K., Mahajan, M. (eds.) IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2010, Chennai, India, December 15-18, 2010, LIPIcs, vol. 8, pp. 412–423. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2010). https://doi.org/10.4230/LIPIcs.FSTTCS.2010.412
13. Bruyère, V., Pérez, G.A., Raskin, J., et al.: Partial solvers for generalized parity games. In: Filiot, E., Jungers, R.M., Potapov, I. (eds.) Reachability Problems – 13th International Conference, RP 2019, Brussels, Belgium, September 11–13, 2019, Proceedings, Lecture Notes in Computer Science, vol. 11674, pp. 63–78. Springer, Berlin (2019). https://doi.org/10.1007/978-3-030-30806-3_6
14. Cadilhac, M., Pérez, G.A.: Acacia-bonsai: a modern implementation of downset-based LTL realizability. In: Tools and Algorithms for the Construction and Analysis of Systems – 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023. Paris, France, April 22-27, 2023, Proceedings, Part II, Lecture Notes in Computer Science, vol. 13994, pp. 192–207. Springer, Berlin (2023).
15. Camacho, A., Muise, C.J., Baier, J.A., et al.: LTL realizability via safety and reachability games. In: IJCAI. ijcai.org pp. 4683–4691 (2018)
16. Casares, A., Colcombet, T., Fijalkow, N.: Optimal transformations of games and automata using Muller conditions. In: Bansal, N., Merelli, E., Worrell, J. (eds.) 48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, Glasgow, Scotland (Virtual Conference), July 12–16, 2021, LIPIcs, vol. 198, pp. 123:1–123:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.ICALP.2021.123
17. Casares, A., Duret-Lutz, A., Meyer, K.J., et al.: Practical applications of the alternating cycle decomposition. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems – 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022. Proceedings, Part II, Lecture Notes in Computer Science, vol. 13244, pp. 99–117. Springer, Berlin (2022). https://doi.org/10.1007/978-3-030-99527-0_6
18. Chatterjee, K., Henzinger, T.A., Piterman, N.: Generalized parity games. In: Seidl, H. (ed.) Foundations of Software Science and Computational Structures, 10th International Conference, FOSSACS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 24–April 1, 2007. Proceedings, Lecture Notes in Computer Science, vol. 4423, pp. 153–167. Springer, Berlin (2007). https://doi.org/10.1007/978-3-540-71389-0_12
19. Church, A.: Application of recursive arithmetic to the problem of circuit synthesis. In: Summaries of the Summer Institute of Symbolic Logic, vol. 1, pp. 3–50. Am. Math. Soc., Providence (1957)
20. Church, A.: Logic, arithmetic, and automata. J. Symb. Log. **29**(4) (1964)
21. Clarke, E.M., Henzinger, T.A., Veith, H., et al. (eds.): Handbook of Model Checking Springer, Berlin (2018). https://doi.org/10.1007/978-3-319-10575-8
22. de Alfaro, L., Roy, P.: Solving games via three-valued abstraction refinement. Inf. Comput. **208**(6), 666–676 (2010). https://doi.org/10.1016/j.ic.2009.05.007
23. De Berg, M., Cheong, O., van Kreveld, M.J., et al.: Computational Geometry: Algorithms and Applications, 3rd edn. Springer, Berlin (2008). https://www.worldcat.org/oclc/227584184
24. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., et al.: Spot 2.0 — a framework for LTL and ω-automata manipulation. In: Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16). Lecture Notes in Computer Science, vol. 9938, pp. 122–129. Springer, Berlin (2016). https://doi.org/10.1007/978-3-319-46520-3_8
25. Ehlers, R.: Symbolic bounded synthesis. In: International Conference on Computer Aided Verification, pp. 365–379. Springer, Berlin (2010)
26. Ehlers, R.: Unbeast: symbolic bounded synthesis. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 272–275. Springer, Berlin (2011)
27. Esparza, J., Křetínský, J., Sickert, S.: One theorem to rule them all: a unified translation of LTL into ω-automata. In: Dawar, A., Grädel, E. (eds.) Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'18), pp. 384–393. ACM, New York (2018). https://doi.org/10.1145/3209108.3209161
28. Esparza, J., Kretínský, J., Sickert, S.: A unified translation of linear temporal logic to ω-automata. J. ACM **67**(6), 33:1–33:61 (2020). https://doi.org/10.1145/3417995
29. Faymonville, P., Finkbeiner, B., Tentrup, L.: Bosy: an experimentation framework for bounded synthesis. In: Majumdar, R., Kuncak, V. (eds.) Computer Aided Verification – 29th International Conference, CAV 2017, Heidelberg, Germany, July 24–28, 2017. Proceedings, Part II, Lecture Notes in Computer Science, vol. 10427, pp. 325–332. Springer, Berlin (2017). https://doi.org/10.1007/978-3-319-63390-9_17
30. Finkbeiner, B., Klein, F.: Bounded cycle synthesis. In: Chaudhuri, S., Farzan, A. (eds.) Proceedings, Part I, Computer Aided Verification – 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17–23, 2016. Lecture Notes in Computer Science, vol. 9779, pp. 118–135. Springer, Berlin (2016). https://doi.org/10.1007/978-3-319-41528-4_7
31. Finkbeiner, B., Klein, F., Piskac, R., et al.: Synthesizing functional reactive programs. In: Eisenberg, R.A. (ed.) Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019, pp. 162–175. ACM, New York (2019). https://doi.org/10.1145/3331545.3342601
32. Finkbeiner, B., Klein, F., Piskac, R., et al.: Temporal stream logic: synthesis beyond the bools. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification – 31st International Conference, CAV 2019, New York City, NY, USA, July 15–18, 2019. Proceedings, Part I, Lecture Notes in Computer Science, vol. 11561,

pp. 609–629. Springer, Berlin (2019). https://doi.org/10.1007/978-3-030-25540-4_35
33. Finkbeiner, B., Geier, G., Passing, N.: Specification decomposition for reactive synthesis. In: Proceedings for the 13th NASA Formal Methods Symposium (NFM'21) (2021). https://doi.org/10.1007/978-3-030-76384-8_8
34. Friedmann, O., Lange, M.: Solving parity games in practice. In: Liu, Z., Ravn, A.P. (eds.) Automated Technology for Verification and Analysis, 7th International Symposium, ATVA 2009, Macao, China, October 14–16, 2009, Proceedings, Lecture Notes in Computer Science, vol. 5799, pp. 182–196. Springer, Berlin (2009). https://doi.org/10.1007/978-3-642-04761-9_15
35. Geier, G., Heim, P., Klein, F., et al.: Syntroids: synthesizing a game for FPGAs using temporal logic specifications. In: Barrett, C.W., Yang, J. (eds.) Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22–25, 2019, pp. 138–146. IEEE, Los Alamitos (2019). https://doi.org/10.23919/FMCAD.2019.8894261
36. Jacobs, S.: Extended AIGER format for synthesis. CoRR (2014). http://arxiv.org/abs/1405.5793. arXiv:1405.5793
37. Jacobs, S., Perez, G.A.: Data and scripts used for the SYNTCOMP report 2018–2021 (2023). https://doi.org/10.5281/zenodo.7588780
38. Jacobs, S., Sakr, M.: AIGEN: random generation of symbolic transition systems. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification – 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12760, pp. 435–446. Springer, Berlin (2021). https://doi.org/10.1007/978-3-030-81688-9_20
39. Jacobs, S., Klein, F., Schirmer, S.: A high-level LTL synthesis format: TLSF v1.1. In: Piskac, R., Dimitrova, R. (eds.) Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17–18, 2016, pp. 112–132 (2016). https://doi.org/10.4204/EPTCS.229.10
40. Jacobs, S., Basset, N., Bloem, R., et al.: The 4th reactive synthesis competition (SYNTCOMP 2017): benchmarks, participants & results. In: Fisman, D., Jacobs, S. (eds.) Proceedings Sixth Workshop on Synthesis, SYNT@CAV 2017, Heidelberg, Germany, 22nd July 2017, pp. 116–143 (2017). https://doi.org/10.4204/EPTCS.260.10
41. Jacobs, S., Bloem, R., Brenguier, R., et al.: The first reactive synthesis competition (SYNTCOMP 2014). Int. J. Softw. Tools Technol. Transf. **19**(3), 367–390 (2017) https://doi.org/10.1007/s10009-016-0416-3
42. Kretínský, J., Meggendorfer, T., Sickert, S.: Owl: a library for ω-words, automata, and LTL. In: Lahiri, S.K., Wang, C. (eds.) Automated Technology for Verification and Analysis – 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7–10, 2018. Proceedings, Lecture Notes in Computer Science, vol. 11138, pp. 543–550. Springer, Berlin (2018). https://doi.org/10.1007/978-3-030-01090-4_34
43. Lijzenga, O., van Dijk, T.: Symbolic parity game solvers that yield winning strategies. In: Raskin, J., Bresolin, D. (eds.) Proceedings 11th International Symposium on Games, Automata, Logics, and Formal Verification, GandALF 2020, Brussels, Belgium, September 21–22, 2020, pp. 18–32 (2020). https://doi.org/10.4204/EPTCS.326.2
44. Luttenberger, M.: Strategy iteration using non-deterministic strategies for solving parity games. CoRR (2008). http://arxiv.org/abs/0806.2923. arXiv:0806.2923
45. Luttenberger, M., Meyer, P.J., Sickert, S.: Practical synthesis of reactive systems from LTL specifications via parity games. Acta Inform. **57**(1–2), 3–36 (2020). https://doi.org/10.1007/s00236-019-00349-3
46. Meyer, P.J., Sickert, S.: Modernising Strix. Presented at the 10th Workshop on Synthesis (2021)
47. Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: explicit reactive synthesis strikes back! In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14–17, 2018. Proceedings, Part I, Lecture Notes in Computer Science, vol. 10981, pp. 578–586. Springer, Berlin (2018). https://doi.org/10.1007/978-3-319-96145-3_31
48. Müller, D., Sickert, S.: LTL to deterministic Emerson–Lei automata. In: Bouyer, P., Orlandini, A., Pietro, P.S. (eds.) Proceedings of the Eighth International Symposium on Games, Automata, Logics and Formal Verification (GandALF'17), pp. 180–194 (2017). https://doi.org/10.4204/EPTCS.256.13
49. Pérez, G.A.: The extended HOA format for synthesis. CoRR (2019). http://arxiv.org/abs/1912.05793. arXiv:1912.05793
50. Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. Log. Methods Comput. Sci. **3**(3) (2007). https://doi.org/10.2168/LMCS-3(3:5)2007
51. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, 31 October–1 November 1977, pp. 46–57. IEEE Comput. Soc., Providence (1977). https://doi.org/10.1109/SFCS.1977.32
52. Renkin, F., Duret-Lutz, A., Pommellet, A.: Practical "paritizing" of Emerson–Lei automata. In: Proceedings of the 18th International Symposium on Automated Technology for Verification and Analysis (ATVA'20). Lecture Notes in Computer Science, vol. 12302, pp. 127–143. Springer, Berlin (2020). https://doi.org/10.1007/978-3-030-59152-6_7
53. Renkin, F., Schlehuber, P., Duret-Lutz, A., et al.: Improvements to *ltlsynt*. Presented at the 10th Workshop on Synthesis (2021) https://hal.archives-ouvertes.fr/hal-03523385
54. Renkin, F., Schlehuber-Caissier, P., Duret-Lutz, A., et al.: Effective reductions of Mealy machines. In: Proceedings of the 42nd International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE'22). Lecture Notes in Computer Science, vol. 13273, pp. 114–130. Springer, Berlin (2022).
55. Safra, S.: On the complexity of omega-automata. In: 29th Annual Symposium on Foundations of Computer Science, White Plains, 24–26 October 1988, pp. 319–327. IEEE Comput. Soc., New York (1988). https://doi.org/10.1109/SFCS.1988.21948
56. Seidl, M., Könighofer, R.: Partial witnesses from preprocessed quantified Boolean formulas. In: Fettweis, G.P., Nebel, W. (eds.) Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24–28, 2014. European Design and Automation Association, pp. 1–6 (2014). https://doi.org/10.7873/DATE.2014.162
57. Sickert, S., Esparza, J.: An efficient normalisation procedure for linear temporal logic and very weak alternating automata. In: Hermanns, H., Zhang, L., Kobayashi, N., et al. (eds.) LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8–11, 2020, pp. 831–844. ACM, New York (2020). https://doi.org/10.1145/3373718.3394743
58. Somenzi, F.: CUDD package, release 2.4.1 (2005). http://vlsi.colorado.edu/~fabio/CUDD/
59. Stump, A., Sutcliffe, G., Tinelli, C.: Starexec: a cross-community infrastructure for logic solving. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) Automated Reasoning – 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19–22, 2014. Proceedings, Lecture Notes in Computer Science, vol. 8562, pp. 367–373. Springer, Berlin (2014). https://doi.org/10.1007/978-3-319-08587-6_28
60. Tentrup, L., Rabe, M.N.: Clausal abstraction for DQBF. In: SAT, Lecture Notes in Computer Science, vol. 11628, pp. 388–405. Springer, Berlin (2019)

61. Van Dijk, T.: Oink: an implementation and evaluation of modern parity game solvers. In: Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'18), pp. 291–308. Springer, Berlin (2018). https://doi.org/10.1007/978-3-319-89960-2_16
62. Van Dijk, T., Rubbens, B.: Simple fixpoint iteration to solve parity games. In: Leroux, J., Raskin, J. (eds.) Proceedings Tenth International Symposium on Games, Automata, Logics, and Formal Verification, GandALF 2019, Bordeaux, France, 2–3 September 2019, pp. 123–139 (2019). https://doi.org/10.4204/EPTCS.305.9
63. 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). https://doi.org/10.1007/s10009-016-0433-2
64. Vardi, M.Y., Wolper, P.: Automata theoretic techniques for modal logics of programs (extended abstract). In: DeMillo, R.A. (ed.) Proceedings of the 16th Annual ACM Symposium on Theory of Computing, Washington, DC, USA, April 30–May 2, 1984, pp. 446–456. ACM, New York (1984). https://doi.org/10.1145/800057.808711