The 4th Reactive Synthesis Competition (SYNTCOMP 2017): Benchmarks, Participants & Results

Swen Jacobs
Saarland University
Saarbrücken, Germany

Nicolas Basset
Université Libre de Bruxelles
Brussels, Belgium

Roderick Bloem
Graz University of Technology
Graz, Austria

Romain Brenguier
University of Oxford
Oxford, UK

Maximilien Colange
LRDE, EPITA
Kremlin-Bicêtre, France

Peter Faymonville
Saarland University
Saarbrücken, Germany

Bernd Finkbeiner
Saarland University
Saarbrücken, Germany

Ayrat Khalimov
Graz University of Technology
Graz, Austria

Felix Klein
Saarland University
Saarbrücken, Germany

Thibaud Michaud
LRDE, EPITA
Kremlin-Bicêtre, France

Guillermo A. Pérez
Université Libre de Bruxelles
Brussels, Belgium

Jean-François Raskin
Université Libre de Bruxelles
Brussels, Belgium

Ocan Sankur
CNRS, Irisa
Rennes, France

Leander Tentrup
Saarland University
Saarbrücken, Germany

We report on the fourth reactive synthesis competition (SYNTCOMP 2017). We introduce two new benchmark classes that have been added to the SYNTCOMP library, and briefly describe the benchmark selection, evaluation scheme and the experimental setup of SYNTCOMP 2017. We present the participants of SYNTCOMP 2017, with a focus on changes with respect to the previous years and on the two completely new tools that have entered the competition. Finally, we present and analyze the results of our experimental evaluation, including a ranking of tools with respect to quantity and quality of solutions.

1 Introduction

The reactive synthesis competition (SYNTCOMP) has been founded in 2014 [22] with the goal to increase the impact of theoretical advancements in the automatic synthesis of reactive systems. Reactive synthesis is one of the major challenges of computer science since the definition of the problem more than 50 years ago [11]. A large body of theoretical results has been developed since then, but their impact on the practice of system design has been rather limited. SYNTCOMP is designed to foster research in scalable and user-friendly implementations of synthesis techniques by establishing a standard benchmark format, maintaining a challenging public benchmark library, and providing a *dedicated and independent* platform for the comparison of tools under consistent experimental conditions.

Since its inception, SYNTCOMP is held annually, and is associated with the International Conference on Computer Aided Verification (CAV) and the Workshop on Synthesis (SYNT), where the competition results are presented to the community. [22–24] A design choice for the first two competitions was to focus on safety properties specified as monitor circuits in an extension of the AIGER format known from the hardware model checking competition [9, 20]. SYNTCOMP 2016 introduced the first major extension of the competition by adding a new track that is based on properties in full linear temporal logic (LTL), given in the *temporal logic synthesis format* (TLSF) [21, 25].

The organization team of SYNTCOMP 2017 consisted of R. Bloem and S. Jacobs.

D. Fisman, S. Jacobs (Eds.): Sixth Workshop on Synthesis (SYNT 2017)
EPTCS 260, 2017, pp. 116–143, doi:10.4204/EPTCS.260.10

© S. Jacobs et al.
This work is licensed under the
Creative Commons Attribution License.

Outline. The rest of this paper describes the design, benchmarks, participants, and results of SYNTCOMP 2017. In Section 2, we present two new benchmark classes that have been added to the SYNTCOMP library and give an overview of the benchmark set for SYNTCOMP 2017. In Section 3, we describe the setup, rules and execution of the competition. In Section 4 we give an overview of the participants of SYNTCOMP 2017, focusing on changes compared to last year's participants. The experimental results are presented and analyzed in Section 5, before we end with some concluding remarks in Section 6.

2 Benchmarks

In this section, we describe the benchmark library for SYNTCOMP 2017. We start by describing two new benchmark classes in TLSF, followed by a listing of the classes of benchmarks (in both TLSF and AIGER format) that have already been used in previous competitions. For more details on these existing benchmarks, we refer to the previous competition reports [22–24].

2.1 New Benchmark Set: Decomposed AMBA

This set of benchmarks has first been presented as an example in the presentation of TLSF [25]. It describes the well-known AMBA bus controller, decomposed into eight components that can be synthesized independently. Out of these eight components, three are parameterized in the number of systems that access the shared bus, providing more challenging synthesis problems for larger parameter values. These benchmarks have been translated to TLSF by F. Klein.

2.2 New Benchmark Set: Unrealizable Variants

This set of benchmarks is based on a number of existing TLSF benchmark classes in the SYNTCOMP library: detector, full arbiter, load balancer, prioritized arbiter, round robin arbiter, simple arbiter (described below). All of the original benchmarks are parameterized in the number of systems that can send requests to the synthesized component. Moreover, all of them include a mutual exclusion property, and in this benchmark set have been modified in one of two ways to obtain unrealizable variants of the respective specification:

1. We add a requirement that the system has to serve multiple requests within a fixed number of steps, resulting in unrealizability due to a clash of mutual exclusion and the time limit on serving the requests.
2. We add a requirement that forces the system to violate mutual exclusion not after a fixed time, but at some undetermined time in the future.

2.3 Existing Benchmarks: TLSF

In addition to the new benchmarks described above, we briefly describe existing TLSF benchmarks. For more details consult the report on SYNTCOMP 2016 [23] and the original sources. The existing benchmark library consists of the following classes of benchmarks:

- **Lily benchmark set:** the set of benchmarks originally included with the LTL synthesis tool LILY [28]. It includes 24 benchmarks.

- **Acacia benchmark set:** the set of benchmarks originally included with the LTL synthesis tool Acacia+ [5]. It includes 65 benchmarks.
- **Parameterized detector:** specifies a component that raises its single output infinitely often if and only if all its inputs are raised infinitely often. Parameterized in the number of inputs.
- **Parameterized arbiters:** four arbiter specifications of different complexity (simple arbiter, prioritized arbiter, round robin arbiter, full arbiter). Parameterized in the number of masters that the arbiter needs to serve.
- **Parameterized AMBA bus controller:** essentially an arbiter with a large number of features, including prioritization and locking of the bus for a fixed or arbitrary number of steps [27]. Parameterized in the number of masters that the controller has to serve.
- **Parameterized load balancer:** a component that receives jobs and distributes them to a fixed number of servers [13]. Parameterized in the number of servers that can handle the jobs.
- **Parameterized generalized buffer:** a family of buffers that transmit data from a number of senders to two receivers, based on a handshake protocol and a FIFO queue that is used to store data [27]. The benchmark is parameterized in the number of senders.
- **Parameterized LTL to Büchi translation (LTL2DBA):** generation of deterministic Büchi automata that correspond to a specification taken from a set of parameterized LTL formulas [41].

2.4 Existing Benchmarks: AIGER

We briefly describe the existing library of AIGER benchmarks. For more details, consult the previous competition reports [22–24] and the original sources.

- **HWMCC benchmarks:** based on a subset of the benchmarks from HWMCC 2012 and HWMCC 2014 [9], where a subset of the inputs have been declared as controllable, and a safe controller for these inputs should be synthesized. This benchmark set contains 390 benchmarks.
- **(Bounded) LTL to Büchi and LTL to parity translation (LTL2DBA/LTL2DPA):** based on the benchmarks for LTL2DBA benchmarks from Section 2.3, but including also the synthesis of parity automata, and additionally parameterized in the liveness-to-safety approximation. 62 instances.
- **Toy Examples:** a number of basic building blocks of circuits, such as an adder, a bitshifter, a counter, and a multiplier. The set consists of 176 problem instances.
- **AMBA:** a version of the bus controller specification for AMBA [27], parameterized in three dimensions (number of masters, type and precision of the liveness-to-safety approximation). The benchmark set contains 952 instances.
- **Genbuf:** a version of the generalized buffer specification [27], parameterized in the same way as the AMBA benchmarks. The set contains 866 instances.
- **LTL2AIG:** several sets of benchmarks that are based on the benchmark set of synthesis tool Acacia+ [5], translated using the LTL2AIG tool [22]. This includes versions of the Lily, generalized buffer, and load balancer benchmarks mentioned above. 197 problem instances.
- **Factory Assembly Line:** a controller for two robot arms on an assembly line. This set contains 15 problem instances.

- **Moving Obstacle Evasion:** a moving robot that should evade a moving obstacle in two-dimensional space. The set consists of 16 problem instances.
- **Washing Cycle Scheduler:** a controller of a washing system, with water tanks that share pipes. Parameterized in the number of tanks, the maximum reaction delay, and the shared water pipes. The set contains 321 instances.
- **Driver Synthesis:** specifies a driver for a hard disk controller with respect to a given operating system model [38]. Parameterized in the level of data abstraction, the precision of the liveness-to-safety approximation, and the simplification of the specification circuit by ABC [6]. 72 instances.
- **Huffman Encoder:** specifies a given Huffman decoder, for which a suitable encoder should be synthesized [29]. Parameterized in the liveness-to-safety approximation, resulting in 5 instances.
- **HyperLTL:** based on benchmark problems from HyperLTL model checking [17]. The goal is to synthesize a witness for a given HyperLTL property. This benchmark set contains 21 instances.
- **Matrix Multiplication:** asks for a circuit that performs a single matrix multiplication, or repeated multiplication with a subset of controllable inputs and an additional safety goal. Parameterized in the size of the input matrices, resulting in 354 problem instances.

3 Setup, Rules and Execution

We give an overview of the setup, rules and execution of SYNTCOMP 2017. More details, and the reasoning behind different design choices, can be found in the first competition report [22] and previous work in which we also outlined plans for future extensions of the competition [21].

3.1 General Rules

Like in the previous year, there are two main tracks: one is based on safety specifications in AIGER format (in the following: AIGER/safety-track), and the other on full LTL specifications in TLSF (in the following: TLSF/LTL-track). The 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). In the following, we start with rules that are common to both tracks, followed by rules that are specific to one of the tracks.

Submissions. Tools are submitted as source code, with instructions for installation and a description of the algorithms and optimizations used to solve the synthesis problem. Every tool can run in up to three configurations per subtrack and execution mode. After the initial submission, every tool is tested on a small set of benchmarks from the SYNTCOMP library, and authors are informed about any problems and can submit bugfixes.[1]

Ranking Schemes. In all tracks, there is a ranking based on the number of correctly solved problems: a correct answer within the timeout of 3600s is rewarded with one point for the solver, and a wrong answer is punished by subtracting 4 points. In the realizability tracks, correctness is determined by

[1] Besides revealing bugs or shortcomings in the participating tools themselves, this year these tests have also revealed a number of problems in tools that are used as subprocedures. All of these problems have subsequently been addressed, thus providing an additional benefit of the competition to the broader community of formal methods research.

the realizability information stored in the files, if they have been used in previous competitions, or on a majority vote of the tools that solve the benchmark, otherwise. In the synthesis tracks, if the specification is realizable, then solution has to be model checked. This differs based on the input format, as explained below.

Furthermore, 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 s of the solution is compared to the size *ref* of a reference solution, which is either the smallest solution obtained by competition tools (during competition runs or special reference runs), or, if no previous solution exists, the smallest solution obtained by a tool in the current competition. The number of points obtained for a correct solution decreases logarithmically in the ratio of s and *ref*, i.e., a correct answer (within the timebound) is rewarded with

$$2 - log_{10}\left(\frac{s+1}{ref+1}\right)$$

points. Roughly, this means that for a solution with the same size as the best known solution, 2 points are awarded. If the new solution is 10 times bigger, 1 point is awarded. If it is 10 times smaller, 3 points are awarded. A solution that is more than 100 times bigger than *ref* is awarded 0 points. Note that since some synthesis problems can be solved without using a single gate, we cannot use the ratio $\frac{s}{ref}$, and use $\frac{s+1}{ref+1}$ instead.

3.2 Specific Rules for AIGER/safety-Track

Input Format. In the AIGER/safety-track, specifications are given in the Extended AIGER Format for Synthesis [20, 22], modeling a single safety property.

Correctness of Solutions. In the synthesis subtrack, if the specification is realizable then tools must produce a solution in AIGER format. For *syntactical correctness*, this solution must include the specification circuit, and must define how the inputs that are declared as `controllable` are computed from the `uncontrollable` inputs and the state variables of the circuit (for details, see the SYNTCOMP 2014 report [22]). To ensure also *semantical correctness*, the solutions are additionally model checked, and only solutions that are both syntactically and semantically verified are counted as correct. To facilitate verification, synthesis tools can optionally output an inductive invariant that witnesses the correctness of the solution, e.g., the winning region resulting from the attractor computation. Such an invariant is used *in addition* to model checking, i.e., if the invariant check is inconclusive we fall back to full model checking.

3.3 Specific Rules for TLSF/LTL-Track

Input Format. In the TLSF/LTL-track, specifications are given in TLSF [25]. The organizers supply the *synthesis format conversion* (SyFCo) tool[2] that can be used to translate the specification to a large number of existing specification formats. Specifications are interpreted according to standard LTL semantics, with respect to realizability as a Mealy machine.

[2]SyFCo is available at `https://github.com/reactive-systems/syfco`. Accessed August 2017.

Correctness of Solutions. In the synthesis subtrack, tools have to produce a solution in AIGER format if the specification is realizable. For *syntactical correctness*, the sets of inputs and outputs of the specification must be identical to the sets of inputs and outputs of the solution. To verify *semantical correctness*, the solutions are additionally model checked against the specification with existing model checking tools. Only a solution that can be verified both syntactically and semantically is counted as correct.

3.4 Selection of Benchmarks

AIGER/safety-track. In the AIGER-based track, the selection of benchmarks is based on information about the realizability and difficulty of benchmark problems that has been obtained from the results of previous competitions. This information is stored inside the benchmark files, as described in the SYNTCOMP 2015 report [24]. For realizable specifications, we additionally determined the smallest known solution, stored as a *reference size*.

In SYNTCOMP 2017, we used the same benchmark classes and the same number of problems per class as in the previous year, but we randomly exchanged some of the problems within any given class (for classes that contain more problems than are selected for the competition), while preserving an even distribution of difficulty over the given class. The number of selected problems from each category (cp. Section 2.4) is given in Table 1.

Table 1: Number of selected Benchmarks per Category, AIGER/safety-track

Category	Benchmarks	Category	Benchmarks
AMBA	16	Genbuf (LTL2AIG)	8
(Washing) Cycle Scheduling	16	Add (Toy Examples)	5
Demo (LTL2AIG)	16	Count (Toy Examples)	5
Driver Synthesis	16	Bitshift (Toy Examples)	5
Factory Assembly Line	15	Mult (Toy Examples)	5
Genbuf	16	Mv/Mvs (Toy Examples)	5
HWMCC	16	Stay (Toy Examples)	5
HyperLTL	16	Huffman Encoder	5
Load Balancer (LTL2AIG)	16		
LTL2DBA/LTL2DPA	16		
Moving Obstacle	16		
Matrix Multiplication	16	**Total:**	**234**

TLSF/LTL-track. In the TLSF-based track, for realizability checking we used all 24 of the non-parameterized benchmarks from the Lily benchmark set, and 64 from the Acacia benchmark set. Additionally, we used 6 instances of each of the parameterized benchmarks. Overall, this amounts to 244 problem instances.

3.5 Execution

Like in the last two years, SYNTCOMP 2017 was run at Saarland University, on a set of identical machines with a single quad-core Intel Xeon processor (E3-1271 v3, 3.6GHz) and 32 GB RAM (PC1600, ECC), running a GNU/Linux system. Each node has a local 480 GB SSD that can be used as temporary storage.

Also like in previous years, the competition was organized on the EDACC platform [3], with a very similar setup. To ensure a high comparability and reproducability of our results, a complete machine was reserved for each job, i.e., one synthesis tool (configuration) running one benchmark. Olivier Roussel's `runsolver` [37] was used to run each job and to measure CPU time and wall time, as well as enforcing timeouts. As all nodes are identical and no other tasks were run in parallel, no other limits than a timeout of 3600 seconds (CPU time in sequential mode, wall time in parallel mode) per benchmark was set. Like last year, we used wrapper scripts to execute solvers that did not conform completely with the output format specified by the competition, e.g., to filter extra information that was displayed in addition to the specified output.

The model checker used for checking correctness of solutions for the AIGER/Safety track is IIMC[3] in version 2.0. For solvers that supply an inductive invariant as a witness of correctness, we used a BDD-based invariant check to check correctness[4], and used full model-checking as a fallback solution if the invariant check failed.

For the TLSF/LTL track, the model checker used was V3[5] [43].

4 Participants

Overall, ten tools were entered into SYNTCOMP 2017: five in the AIGER/safety-track, and five in the TLSF/LTL-track. We briefly describe the participants and give pointers to additional information.

4.1 AIGER/safety-Track

This track had five participants in 2017, which we briefly describe in the following. For additional details on the implemented techniques and optimizations, we refer to the previous SYNTCOMP reports [22–24].

Updated Tool: Swiss AbsSynthe v2.1

AbsSynthe was submitted by R. Brenguier, G. A. Pérez, J.-F. Raskin, and O. Sankur, and competed in both the realizability and the synthesis track. It implements the classical backward-fixpoint-based approach to solving safety games using BDDs. As additional features, it supports decomposition of the problem into independent sub-games, as well as an abstraction approach [7, 8]. This year, AbsSynthe contains a new approach to compositionality, where the problem is not separated into as many sub-games as possible, but rather merges some of the smaller sub-games in the hope that this will yield more useful information about the overall problem. It competes in the following sequential (SCx) and parallel (PCx) configurations:

---

[3] IIMC is available at `ftp://vlsi.colorado.edu/pub/iimc/`. Accessed August 2017.

[4] Available from the SYNTCOMP repository at `https://bitbucket.org/swenjacobs/syntcomp/src`, in subdirectory `tools/WinRegCheck`. Accessed August 2017.

[5] V3 is available at `https://github.com/chengyinwu/V3`. Accessed August 2017.

- (SC1) uses a standard BDD-based fixpoint computation with several optimizations, but without compositionality or abstraction,
- (SC2) uses an abstraction algorithm, but no compositionality, and
- (SC3) uses a compositional algorithm, combined with an abstraction method. This is the only sequential configuration that changed this year, incorporating the new approach to compositionality mentioned above.
- (PC1) runs the three sequential configurations in parallel, plus one additional configuration that uses abstraction with fixed threshold, but no compositionality,
- (PC2) runs four copies of (PC1), only modified in the BDD reordering technique, and
- (PC3) runs four copies of (PC2), with the same set of different reordering techniques.

Implementation, Availability The source code of AbsSynthe is is available at `https://github.com/gaperez64/AbsSynthe/tree/native-dev-par`.

Re-entered: Demiurge 1.2.0

Demiurge was submitted by R. Könighofer and M. Seidl, and competed in both the realizability and the synthesis track. Demiurge implements different SAT-based methods for solving the reactive synthesis problem [4, 39]. In the competition, three of these methods are used — one of them as the only method in sequential configuration (D1real) , and a combination of all three methods in parallel configuration (P3real). This year, Demiurge competed in the same version as last two years.

Implementation, Availability The source code of Demiurge is available at `https://www.iaik.tugraz.at/content/research/opensource/demiurge/` under the GNU LGPL version 3.

Re-entered: SafetySynth

SafetySynth was submitted by L. Tentrup, and competed in both the realizability and the synthesis track. SafetySynth is a re-implementation of Realizer that implements the standard BDD-based algorithm for safety games, using the optimizations that were most beneficial for BDD-based tools in SYNTCOMP 2014 and 2015 [22, 24]. It competed in the same version as in the previous year, with configurations (basic) and (alternative) that only differ in the BDD reordering heuristic.

Implementation, Availability The source code of SafetySynth is available online at: `https://www.react.uni-saarland.de/tools/safetysynth/`.

Re-entered: Simple BDD Solver

Simple BDD Solver was submitted by L. Ryzhyk and A. Walker, and competed in the realizability track. Simple BDD Solver implements the standard BDD-based algorithm for safety games, including a large number of optimizations in configuration (basic). The other two configurations additionally implement an abstraction-refinement approach inspired by de Alfaro and Roy [1] in two variants: with overapproximation of the winning region in configuration (abs1), or with both over- and underapproximation in (abs2). The version entered into SYNTCOMP 2017 is the same as last year.

Implementation, Availability The source code of Simple BDD Solver is available online at `https://github.com/adamwalker/syntcomp`.

Re-entered: TermiteSAT

TermiteSAT was submitted by A. Legg, N. Narodytska and L. Ryzhyk, and competed in the realizability track. TermiteSAT implements a novel SAT-based method for synthesis of safety specifications based on Craig interpolation. The only configuration in sequential mode implements this new approach, and the parallel configurations (portfolio) and (hybrid) run the new algorithm alongside one of the algorithms of Simple BDD Solver [32], where in (hybrid) there is even communication of intermediate results between the different algorithms.

Implementation, Availability The source code of TermiteSAT is available online at: `https://github.com/alexlegg/TermiteSAT`.

4.2 TLSF/LTL-Track

This track had five participants in 2017, two of which have not participated in previous competitions. All tools competed in both the realizability and the synthesis track. We describe the implemented techniques and optimizations of the new tools, followed by a brief overview of the updated tools. For additional details on the latter, we refer to the report of SYNTCOMP 2016 [23].

New Entrant: BoWSer

BoWSer was submitted by B. Finkbeiner and F. Klein. It implements different extensions of the bounded synthesis approach [18] that solves the LTL synthesis problem by first translating the specification into a universal co-Büchi automaton, and then encoding acceptance of a transition system with bounded number of states into a constraint system. In this case, the constraints are encoded into propositional logic, i.e., a SAT problem. A solution to this SAT problem, i.e., a satisfying assignment, then represents a transition system that satisfies the original specification. To also check for unrealizability of a formula, the dual problem of whether there exists a winning strategy for the environment is also encoded into SAT.

For the synthesis of solutions in the basic configuration the satisfying assignment from the SAT solver is encoded into an AIGER circuit, and then handed to Yosis for simplification. As a first extension, BoWSer implements *bounded cycle synthesis* [16], which restricts the structure of the solution with respect to the number of cycles in the transition system. To this end, it additionally encodes into SAT the existence of a witness structure that guarantees that the number of cycles in the implementation is smaller than a given bound (according to the approach of Tiernan [42]).

In addition, BoWSer supports another encoding into SAT that targets AIGER circuits more directly, where the numbers of gates and latches can be bounded independently.

BoWSer competed in the following configurations:

- configuration (c0) implements bounded synthesis in the basic version mentioned above,
- configuration (c1) implements bounded cycle synthesis on top of bounded synthesis, i.e., in a first step it searches for a solution with a bounded number of states, and if that exists, it additionally bounds the number of cycles
- configuration (c2) also implements bounded cycle synthesis on top of bounded synthesis, with the additional direct encoding into bounded AIGER circuits mentioned above.

In sequential mode, these configurations spawn multiple threads that are executed on a single CPU core. The parallel configurations are essentially identical, except that the threads are distributed to multiple cores.

Implementation, Availability BoWSer is implemented in Haskell, and uses LTL3BA[6] [2] to convert specifications into automata, and MapleSAT[7] [33] to solve SAT queries. For circuit generation, it uses the Yosis framework[8] [19]. The website of BoWSer is `https://www.react.uni-saarland.de/tools/bowser/`, where the source code will be made available soon.

New Entrant: `ltlsynt`

`ltlsynt` was submitted by M. Colange and T. Michaud and competed in a single configuration in both the sequential realizability and sequential synthesis tracks.

To solve the synthesis problem, `ltlsynt` uses a translation to parity games. As a first step, the input LTL formula is translated into an ω-automaton with a transition-based generalized Büchi acceptance condition. The resulting automata are more concise than classical state-based Büchi automata, which is important to make subsequent steps more efficient. Then, the automaton is simplified according to several heuristics, for example by removing non-accepting strongly connected components or redundant acceptance marks. [12] To separate the actions of the environment and the controller, each transition of the obtained automaton is split in two consecutive transitions, corresponding to the uncontrollable inputs and the controllable inputs of the original transition, respectively. The (non-deterministic) split automaton is then translated into a deterministic parity automaton, which can be interpreted as a turn-based parity game that is equivalent to the original synthesis problem. Determinism is key in preserving the semantics of the synthesis problem: every action of the environment can be answered by the controller so that the resulting run satisfies the LTL specification. Here, the controller wins the parity game (recall that such games are determined [34]) if and only if the original instance of the reactive synthesis problem has a solution.

`ltlsynt` implements two algorithms that solve such parity games: the well-known recursive algorithm by Zielonka [44], and the recent quasi-polynomial time algorithm by Calude et al. [10]. Note that the parity automata (and hence the parity games) produced by Spot are transition-based: priorities label transitions, not states. Again, this allows more concise automata, but required to adapt both algorithms to fit this class of automata. The default algorithm of `ltlsynt` is Zielonka's, since in preliminary experiments it outperformed the algorithm of Calude on the benchmarks of SYNTCOMP 2016 [23]. In fact, the experiments also showed that the bottleneck of the procedure is the determinization step, and not the resolution of the parity game.

A winning strategy for the controller in the parity game defines a satisfying implementation of the controller in the synthesis problem. Since parity games are memoryless, or more precisely positional, a winning strategy can be obtained by removing edges of the parity game, so that each controlled state has exactly one outgoing edge. After reversing the splitting operation by merging consecutive transitions in this strategy, it can be straightforwardly encoded into an AIGER circuit. Binary Decision Diagrams (BDDs) are used to represent sets of atomic proposition, allowing some simplifications in the expression of outputs and latches. `ltlsynt` also uses BDDs to cache the expressions represented by AIGER vari-

[6]LTL3BA is available at `https://sourceforge.net/projects/ltl3ba/`. Accessed August 2017.
[7]MapleSAT is available at `https://sites.google.com/a/gsd.uwaterloo.ca/maplesat/`. Accessed August 2017.
[8]Yosis is available at `http://www.clifford.at/yosys/`. Accessed August 2017.

ables to avoid adding redundant gates. In contrast to its competitors, `ltlsynt` does not use external tools such as ABC or Yosis for the encoding of solutions into AIGER.

Implementation, Availability `ltlsynt` is implemented in C++ and integrated into a branch of the Spot automata library [12], which is used for translation of the specification into automata, and for manipulation of automata. Spot also integrates the BDD library BuDDy and the SAT solver PicoSAT. The source code of `ltlsynt` is available in branch `tm/ltlsynt-pg` of the GIT repository of Spot at `https://gitlab.lrde.epita.fr/spot/spot.git`.

Updated Tool: Acacia4Aiger

Acacia4Aiger was submitted by R. Brenguier, G. A. Pérez, J.-F. Raskin, and O. Sankur. It is an extension of the reactive synthesis tool Acacia+[9], which solves the reactive synthesis problem for LTL specifications by a reduction to safety games, which are then solved efficiently by symbolic algorithms based on an *antichain* representation of the sets of states [5]. For SYNTCOMP 2017, Acacia+ has been extended with a parallel mode that searches independently for a system implementation and a counter-strategy for the environment.

Implementation, Availability Acacia4Aiger is implemented in Python and C. It uses the AaPAL library[10] for the manipulation of antichains, and the Speculoos tools [11] to generate AIGER circuits. The source code of Acacia4Aiger is available online at: `https://github.com/gaperez64/acacia4aiger`.

Updated Tool: BoSy

BoSy was submitted by P. Faymonville, B. Finkbeiner and L. Tentrup, and competed in both the realizability and the synthesis track. BoSy uses the *bounded synthesis* approach [18] with an encoding into quantified Boolean formulas (QBF), as described by Faymonville et al. [14,15]. To detect unrealizability, the existence of a bounded strategy of the environment to falsify the specification is encoded in a similar way, and checked in parallel. If no solution is found for a given bound on the size of the implementation, then the bound is increased in an exponential way.

The resulting QBF formulas are first simplified, using the QBF preprocessor bloqqer[12]. In realizability mode, the simplified formula is then directly solved, using the QBF solver RAReQS [26]. In synthesis mode, BoSy uses a combination of RAReQS the certifying QBF solver QuAbS [40], and the certificate returned by QuAbS represents a solution to the synthesis problem. This solution is then converted into AIGER format, and further simplified using the ABC framework.

Two configurations of BoSy competed in SYNTCOMP 2017, differing in the translation from LTL to automata: configuration (spot) uses the Spot framework [12], configuration (ltl3ba) uses LTL3BA [2] for this task. Both configurations support a parallel mode, if more than one core is available. BoSy supports both Mealy and Moore semantics natively, as well as the extraction of a winning strategy for the environment in case of unrealizable specifications.

---

[9] Acacia+ is available at `http://lit2.ulb.ac.be/acaciaplus/`. Accessed August 2017.
[10] AaPAL is available at `http://lit2.ulb.ac.be/aapal/`. Accessed August 2017.
[11] Speculoos is available at `https://github.com/romainbrenguier/Speculoos`. Accessed August 2017.
[12] Bloqqer is available at `http://fmv.jku.at/bloqqer`. Accessed August 2017.

Implementation, Availability. BoSy is written in Swift. It uses LTL3BA or Spot[13] to convert LTL specifications into Büchi automata. It uses bloqqer, RAReQS[14] and QuAbs[15] to solve QBF constraints, and ABC to simplify solutions.

The code is available online at: `https://www.react.uni-saarland.de/tools/bosy/`.

Updated Tool: PARTY

PARTY was submitted by A. Khalimov, and competed in both the realizability and the synthesis track, with three configurations (int, bool, aiger) in sequential mode and one configuration (portfolio) in parallel mode. The tool and the basic algorithms and optimizations it implements have been described in more detail by Khalimov et al. [30, 31].

PARTY uses the bounded synthesis approach [18] for solving the LTL synthesis problem. To detect unrealizability, it uses the standard approach of synthesizing the dualized specification, where the synthesizer searches for a strategy for the environment to falsify the specification. On a given benchmark, PARTY starts two threads to check realizability and unrealizability. The check for unrealizability is limited to 1200 seconds, giving more resources to the realizability check.

PARTY competed in four configurations:

- PARTY (int) is a re-entry from the previous year, with minor updates. In this configuration, the synthesis problem is encoded into SMT satisfiability as follows. The LTL formula is first translated into a universal co-Büchi automaton (UCA) using Spot [12]. Then the emptiness of the product of the automaton and an unknown fixed-size system is encoded into SMT satisfiability. If the SMT query (in logic QF_UFLRA) is satisfiable, then PARTY (int) extracts the state machine, encodes it into Verilog, and translates the Verilog file into AIGER using Yosis. If the SMT query is unsatisfiable, it increases the system size and repeats the previous steps.
- In PARTY (bool) a given LTL formula is translated into a UCA, similar to the approach of PARTY (int). Then, in contrast, PARTY (bool) translates the UCA into a *k-liveness automaton*, where the number of visits to any final state of the original UCA is limited by a heuristically chosen bound. I.e., such an automaton approximates liveness properties up to some bound. The rest is as before, we only note that the resulting SMT query is in the simpler logic QF_UF.
- PARTY (aiger) is a new entrant this year. Similarly to the two previous configurations, it follows the bounded synthesis approach [18], but the games-based one, not the SMT-based one. First, a given LTL formula is reduced into a k-liveness automaton as before. Since the result is a safety automaton, it can be determinized, and translated into a safety game that is then encoded in Verilog. (We use the basic subset construction for determinization, where for each automaton state we introduce one memory latch.) Such a Verilog circuit is then converted into AIGER using Yosis (this translation takes *a lot* of time, likely due to optimizations that Yosis applies). The resulting AIGER synthesis problem is solved with the SDF solver that participated in SYNTCOMP 2016 [23]. SDF was slightly modified to produce AIGER circuits adhering to the TLSF track requirements. If SDF does not find the solution, we increase the parameter k and repeat.
- PARTY (portfolio) runs 5 solvers in parallel: (1) PARTY (aiger), (2) PARTY (aiger) with formula strengthening, (3) PARTY (aiger) on dualized specifications, (4) PARTY (int) with system sizes

[13] Spot is available at `https://spot.lrde.epita.fr`. Accessed August 2017.
[14] RAReQS is available at `http://sat.inesc-id.pt/~mikolas/sw/areqs/`. Accessed August 2017.
[15] QuAbs is available at `https://www.react.uni-saarland.de/tools/quabs/`. Accessed August 2017.

from 1 to 8 (motivation: specifications requiring more than 8 states are unsolvable anyway), and (5) PARTY (bool) with fixed system size of 16 (motivation: solving many unsatisfiable queries for small system sizes takes a significant time). PARTY (portfolio) reports the first solution that any of the solvers find. The choice of these five solvers was based on the known strengths and weaknesses of the individual algorithms, and not on an analysis of their performance (individually or as portfolio) on the SYNTCOMP benchmark set.

Implementation, Availability. PARTY is written in Python. It uses Spot to convert LTL specifications into Büchi automata, Z3[16] [35] to solve SMT constraints, and Yosis to translate Verilog into AIGER. The code is available online at: `https://github.com/5nizza/party-elli`, branch syntcomp17.

5 Experimental Results

We present the results of SYNTCOMP 2017, separated into the AIGER/safety-track and the TLSF/LTL-track. Both main tracks are separated into realizability and synthesis subtracks, and parallel and sequential execution modes. Detailed results of the competition are also directly accessible via the web-frontend of our instance of the EDACC platform at `http://syntcomp.cs.uni-saarland.de`.

5.1 AIGER/safety-Track: Realizability

In the track for realizability checking of safety specifications in AIGER format, 5 tools competed on 234 benchmark instances, selected from the different benchmark categories as explained in Section 3.4. Overall, 16 different configurations entered this track, with 10 using sequential execution mode and 6 using parallel mode. In the following, we compare the results of these 16 configurations on the 234 benchmarks selected for SYNTCOMP 2017.

We first restrict the evaluation of results to purely sequential tools, then extend it to include also the parallel versions, and finally give a brief analysis of the results.

Sequential Mode. In sequential mode, AbsSynthe competed with three configurations (seq1, seq2 and seq3), Demiurge with one configuration (D1real), Simple BDD Solver with three configurations (basic, abs1, abs2), SafetySynth with two configurations (basic and alternative), as well as TermiteSAT with one configuration.

The number of solved instances per configuration, as well as the number of uniquely solved instances, are given in Table 2. No tool could solve more than 171 out of the 234 instances, or about 73% of the benchmark set. 22 instances could not be solved by any tool within the timeout.

Figure 1 gives a cactus plot for the runtimes of the best sequential configuration of each tool.

Parallel Mode. Three of the tools that entered the competition had parallel configurations for the realizability track: three configurations of AbsSynthe (par1, par2, par3), one configuration of Demiurge (P3real), and two configurations of TermiteSAT (portfolio, hybrid). These parallel configurations had to solve the same set of benchmark instances as in the sequential mode. In contrast to the sequential mode, runtime of tools is measured in wall time instead of CPU time. The results are given in Table 3. Compared to sequential mode, a number of additional instances could be solved: both AbsSynthe and

[16] Z3 is available at `https://github.com/Z3Prover/z3`. Accessed August 2017.

Table 2: Results: AIGER Realizability (sequential mode only)

Tool	(configuration)	Solved	Unique
Simple BDD Solver	(abs1)	171	0
SafetySynth	(basic)	165	1
SafetySynth	(alternative)	165	0
Simple BDD Solver	(basic)	165	0
Simple BDD Solver	(abs2)	165	0
AbsSynthe	(SC3)	160	3
AbsSynthe	(SC2)	156	0
AbsSynthe	(SC1)	148	0
Demiurge	(D1real)	127	11
TermiteSAT		101	6

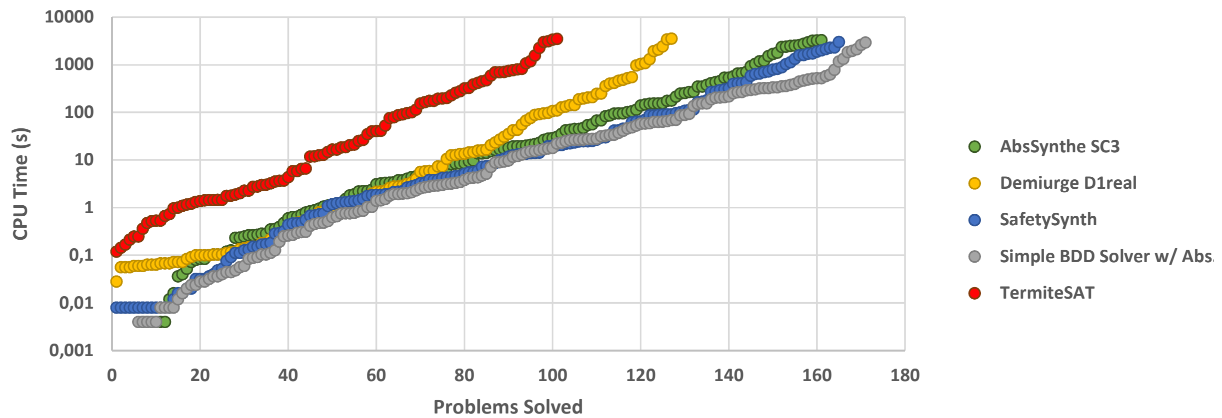

Figure 1: Runtime Cactus Plot of Best Sequential Configurations

TermiteSAT have one or more configurations that solve more then the best tool in sequential mode (about 79% of the benchmark set). Only 15 instances could not be solved by any tool in either sequential or parallel mode.

Note that in Table 3 we only count a benchmark instance as uniquely solved if it is not solved by any other configuration, including the sequential configurations. Consequently, only Demiurge (P3real) produces a single unique solution.

Both modes: Solved Instances by Category. Figure 3 gives an overview of the number of solved instances per configuration and category, for the best sequential and parallel configuration of each tool and the categories defined in Table 1.

Analysis. The best sequential configuration on this year's benchmark set is Simple BDD Solver (abs1), the same as last year. This is the configuration with the simpler form of abstraction. Second place is shared between the other configurations of Simple BDD Solver and the two configurations of Safe-

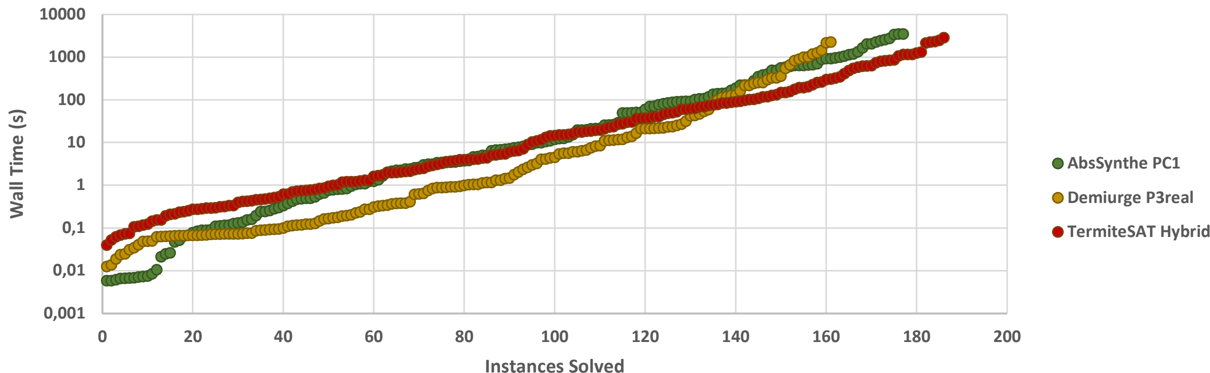

Figure 2: Runtime Cactus Plot of Best Parallel Configurations

Table 3: Results: AIGER Realizability (parallel mode only)

Tool	(configuration)	Solved	Unique
TermiteSAT	(hybrid)	186	0
TermiteSAT	(portfolio)	185	0
AbsSynthe	(PC1)	177	0
Demiurge	(P3real)	161	1
AbsSynthe	(PC3)	156	0
AbsSynthe	(PC2)	147	0

tySynth. Notably, SafetySynth (basic) is the only configuration that solves `amba16f110y`, one of the most difficult (unrealizable) benchmarks from the AMBA class.

These are followed by the three configurations of AbsSynthe, where configuration (SC3), with compositionality and abstraction, performs best and solves 3 benchmarks uniquely (`genbuf64f100y`, `mult_bool_matrix_6_6_8`, and `mult_bool_matrix_6_7_7`), followed by configuration (SC2) which only uses abstraction, and then (SC1) which uses neither compositionality nor abstraction. Like last year, AbsSynthe still uses CUDD v2.5.1, compared to v3.0.0 used in Simple BDD Solver and SafetySynth. We observed last year that the switch to the newer version gave a significant speed-up to Simple BDD Solver.

Finally, Demiurge and TermiteSAT again solve less problems than the BDD-based approaches, but solve a relatively large number of problems uniquely — Demiurge mainly in classes HWMCC, Load Balancer, LTL2DBA/DPA, gb (LTL2AIG) and Huffman, and TermiteSAT mainly in HWMCC, HyperLTL and stay.

Among the parallel configuration, this year the two configurations of TermiteSAT solve most instances (with hybrid mode again only solving one additional instance compared to portfolio mode), followed by last year's winner AbsSynthe (PC1) and the parallel configuration Demiurge (P3real). All of these show that a well-chosen portfolio can solve a significantly higher number of problems than the individual algorithms they are based on. Finally, the parallel configurations (PC2) and (PC3) of AbsSynthe, which represent portfolios of (SC1) and (SC2), respectively, that only differ in the BDD reordering, solve roughly the same number as their sequential counterparts.

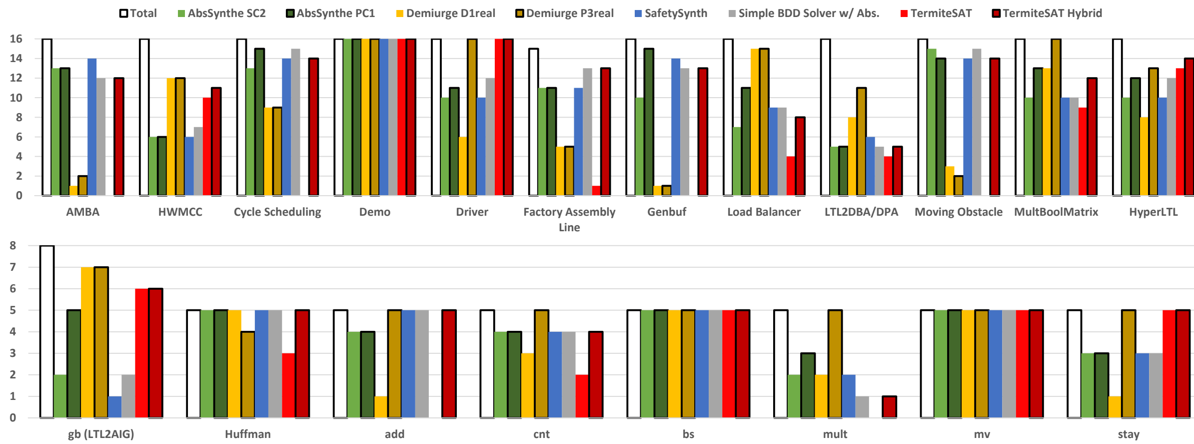

Figure 3: AIGER/safety Realizability Track, Solved Instances by Category

5.2 AIGER/safety-Track: Synthesis

In the track for synthesis from safety specifications in AIGER format, participants had to solve the same set of benchmarks as in the realizability track. Three tools entered the track: AbsSynthe, Demiurge and SafetySynth, in the same configurations as in the realizability track, except for additional synthesis of solutions.

For SYNTCOMP 2017, we have two different rankings in the synthesis track: one is based on the number of instances that can be solved within the timeout, and the other gives a weight to solutions of realizable specifications, based on their size. Furthermore, a solution for a realizable specification is only considered as correct if it can be model-checked within a separate timeout of one hour (cf. Section 3). As before, we first present the results for the sequential configurations, followed by parallel configurations, and end with an analysis of the results.

Sequential Mode. Table 4 summarizes the experimental results, including the number of solved benchmarks, the uniquely solved instances, the number of solutions that could not be model-checked within the timeout, and the accumulated quality of solutions. Note that the "solved" column gives the number of problems that have either been correctly determined unrealizable, or for which the tool has presented a solution that could be verified. With this requirement, no sequential configuration could solve more than 155 or about 66% of the benchmarks, and 48 instances could not be solved by any tool. Very few potential solutions could not be model-checked within the timeout. None of the tools produced any wrong solutions.[17]

Parallel Mode. Table 5 summarizes the experimental results, again including the number of solved benchmarks, the uniquely solved instances, the number of solutions that could not be verified within the timeout, and the accumulated quality of solutions. No tool solved more than 169 problem instances, or about 72% of the benchmark set. Again, there are only very few (potential) solutions that could not be verified within the timeout. None of the solutions have been identified as wrong by our verifiers.

[17] In the EDACC system, benchmark `amba16c40y` is reported with `model checking failed` for configuration AbsSynthe (SC1). A closer inspection showed that this is not due to a faulty solution, but rather due to an uncaught exception in the model checker. Therefore, this solution is neither counted as an incorrect, nor as a correct solution.

Table 4: Results: AIGER Synthesis (sequential mode only)

Tool	(configuration)	Solved	Unique	MC Timeout	Quality
SafetySynth	(basic)	155	2	0	**236**
SafetySynth	(alt)	152	1	0	232
AbsSynthe	(SC2)	149	0	1	191
AbsSynthe	(SC3)	147	0	1	195
AbsSynthe	(SC1)	141	2	0	183
Demiurge	(D1synt)	118	20	1	175

Like in the parallel realizability track, we only consider instances as uniquely solved if they are not solved by any other configuration, including sequential ones. Consequently, none of the configurations produced any unique solutions.

Table 5: Results: AIGER Synthesis (parallel mode only)

Tool	(configuration)	Solved	Unique	MC Timeout	Quality
AbsSynthe	(PC1)	169	0	2	210
Demiurge	(P3Synt)	158	0	0	**266**
AbsSynthe	(PC3)	148	0	2	198
AbsSynthe	(PC2)	139	0	1	179

Analysis. Unsurprisingly, the number of solved instances for each tool in the synthesis track corresponds roughly to those solved in the realizability track. With respect to the smaller number of participants, Demiurge (D1synt) provides an additional 20 unique solutions in the sequential mode, and a few unique solutions are also provided by AbsSynthe (SC1) and SafetySynth (basic and alt).

In sequential mode, the tool with the overall highest quality of solutions is the one which provides also the highest number of solutions, SafetySynth (basic). Note also that here AbsSynthe (SC2) solves more problems than (SC3), but (SC3) has a higher quality score, which means that on average it gives better solutions than (SC2). In parallel mode, the quality score of Demiurge (P3synt) is about 25% higher than for AbsSynthe (PC1), even though the latter solves about 7% more problems. This means that on average the solutions of Demiurge are significantly smaller than those produced by AbsSynthe and SafetySynth. This can be seen in Figure 4, which plots the sizes of synthesized strategies for some of the configurations. We consider here not the size of the complete solution (which includes the specification circuit), but only the number of additional AND-gates, which corresponds to the strategy of the controller.

5.3 TLSF/LTL-Track: Realizability

In the track for realizability checking of LTL specifications in TLSF, 5 tools competed in 8 sequential and 5 parallel configurations. In the following, we compare the results of these 13 configurations on the 244 benchmarks that were selected for SYNTCOMP 2017, as explained in Section 3.4.

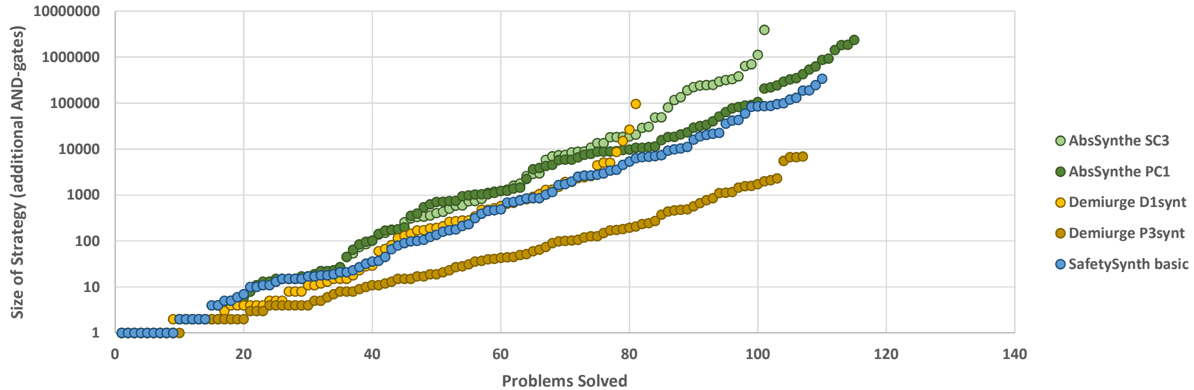

Figure 4: AIGER/safety Synthesis Track: Size of Solution Strategies for Selected Configurations

Again, we first restrict our evaluation to sequential configurations, then extend it to include parallel configurations, and finally give a brief analysis.

Sequential Mode. In sequential mode, Acacia4Aiger, BoWSer and `ltlsynt` each competed with one configuration[18], BoSy with two configurations (ltl3ba and spot), and PARTY with three configurations.

The number of solved instances per configuration, as well as the number of uniquely solved instances, are given in Table 6. No tool could solve more than 218 out of the 244 instances, or about 89% of the benchmark set[19] 14 instances could not be solved by any of the participants within the timeout.

Table 6: Results: TLSF Realizability (sequential mode only)

Tool	(configuration)	Solved	Unique
PARTY	(aiger)	218	7
`ltlsynt`		195	3
BoSy	(spot)	181	0
BoSy	(ltl3ba)	172	0
PARTY	(int)	169	0
BoWSer		165	0
PARTY	(bool)	164	0
Acacia4Aiger		142	4

Figure 5 gives a cactus plot of the runtimes for all sequential algorithms in the realizability track.

Parallel Mode. All tools except `ltlsynt` also entered in one or more parallel configurations: one configuration for each of Acacia4Aiger, BoWSer and party, and two configurations for BoSy. As before,

[18] In fact, BoWSer was entered also in the realizability track in three configurations, but these configurations differ only in the synthesis step. Therefore, they all produced identical results, and are represented as a single configuration here.

[19] For two configurations of PARTY, the numbers reported here differ from those given by the EDACC system. This is because configurations (bool) and (aiger) give result `unrealizable` on benchmarks `ltl2dba_R_10` and `ltl2dba_R_12` only after an uncaught error that is highlighted by the solver output. Therefore, these two (correct) results are not counted here.

Table 7: Results: TLSF Realizability (parallel mode only)

Tool	(configuration)	Solved	Unique
Party	(portfolio)	224	2
BoSy	(spot,par)	181	0
BoWSer	(par)	173	0
BoSy	(ltl3ba,par)	170	0
Acacia4Aiger	(par)	153	0

parallel configurations solve the same set of benchmark instances as in the sequential mode, but runtime is measured in wall time instead of CPU time. The results are given in Table 7. The best parallel configuration solves 224 out of the 244 instances, or about 92% of the benchmark set. 10 benchmarks have not been solved by any configuration.

In Table 7, we again only count a benchmark instance as uniquely solved if it is not solved by any other sequential or parallel configuration. Then, Party (portfolio) solves 2 instances that no other configuration can solve.

Figure 6 gives a cactus plot of the runtimes for the parallel and a selection of the sequential algorithms in the realizability track.

Both modes: Solved Instances of Parameterized Benchmarks. For both the sequential and the parallel configurations, Figure 7 gives an overview of the number of solved instances per configuration, for the 25 parameterized benchmarks used in SYNTCOMP 2017.

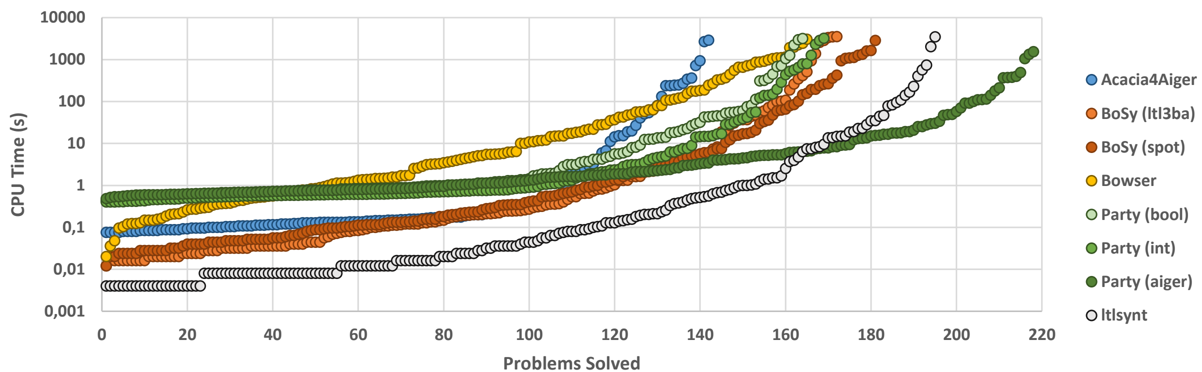

Figure 5: TLSF/LTL Realizability Track: Runtimes of Sequential Configurations

Analysis. In contrast to last year, where the competitors essentially only implemented two different synthesis approaches, the entrants of SYNTCOMP 2017 are much more diverse: BoSy, BoWSer and Party all implement variants of bounded synthesis by encoding into a constraint satisfaction problem, and four other approaches are implemented in Acacia4Aiger, `ltlsynt`, and Party (bool and aiger).[20]

[20]Note that Acacia4Aiger had some technical problems with the syntax of a subset of the new benchmarks for this year, so the number of solved instances does not necessarily reflect the number of instances that could be solved with the implemented

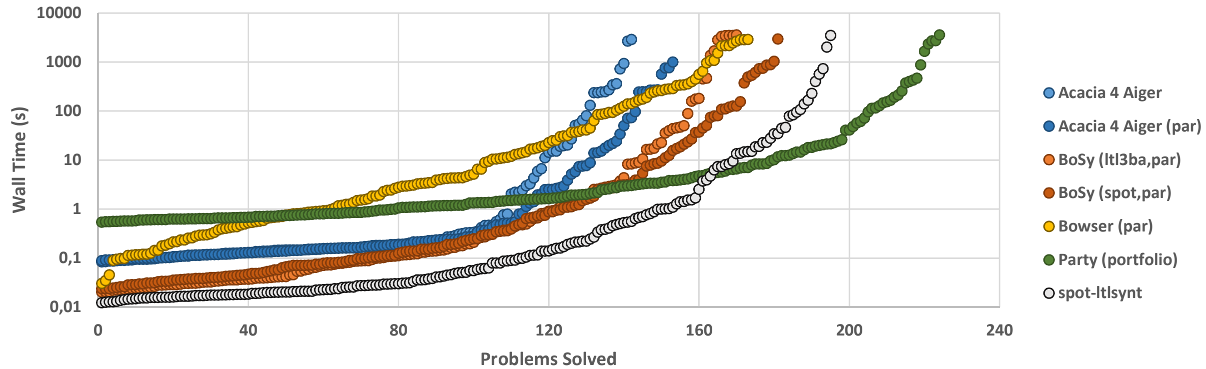

Figure 6: TLSF/LTL Realizability Track: Runtimes of Parallel and Selected Sequential Configurations

Remarkable are the two completely new approaches implemented in PARTY (aiger) and `ltlsynt`, which both solve more problems than the best tools from last year. PARTY (aiger) achieves this by a translation to safety games which approximates liveness properties by bounded *k*-liveness. In contrast to all other approaches, `ltlsynt` does not work by a conversion to a bounded liveness or bounded size problem, which allows the other approaches to avoid determinization of specification automata. Instead, `ltlsynt` achieves very good results with an algorithm based on deterministic automata and parity games, which was widely assumed to be impractical before. We conjecture that this relies on at least two factors: i) very efficient translation and determinization algorithms implemented in Spot [12, 36], and ii) the choice of automata with transition-based acceptance, because their conciseness is key to the efficiency of the overall algorithm.

An analysis of the parameterized benchmarks in Figure 7 shows the different strengths of the approaches: Acacia4Aiger dominates on `generalized_buffer` and `ltl2dba_E`, the bounded synthesis-based approaches on `ltl2dba_R` and `simple_arbiter_unreal2`, the (aiger) configuration of PARTY on `amba_decompsed_lock`, `detector`, `full_arbiter`, `ltl2dba_C2` and `ltl2dba_U1`, and `ltlsynt` on `round_robin_arbiter` and `simple_arbiter_unreal1_4`. For the other parameterized benchmarks, several approaches share the top spot, with the (aiger) configuration of PARTY almost always among the best.

For the benchmarks that have been made unrealizable by adding additional requirements, we also note that the different differences between the approaches: compared to the realizable version `simple_arbiter`, the first unrealizable version `simple_arbiter_unreal1_4` is comparably hard for most tools and easier for some. In contrast, the second unrealizable version `simple_arbiter_unreal2` has a different behavior for most tools, being either significantly harder or significantly easier.

Finally, we note that the two instances that are solved uniquely by PARTY (portfolio) (`prioritized_arbiter_7.tlsf` and `simple_arbiter_12.tlsf`) are solved by the second portfolio solver with formula strengthening, which did not compete in the sequential mode.

5.4 TLSF/LTL-Track: Synthesis

In the track for synthesis from LTL specifications in TLSF, participants had to solve the same benchmarks as in the LTL/TLSF-realizability track. Up to the BoWSer tool, the track had the same participants as the LTL/TLSF-realizability track: Acacia4Aiger with two configurations, BoSy with four configura-

approach in principle.

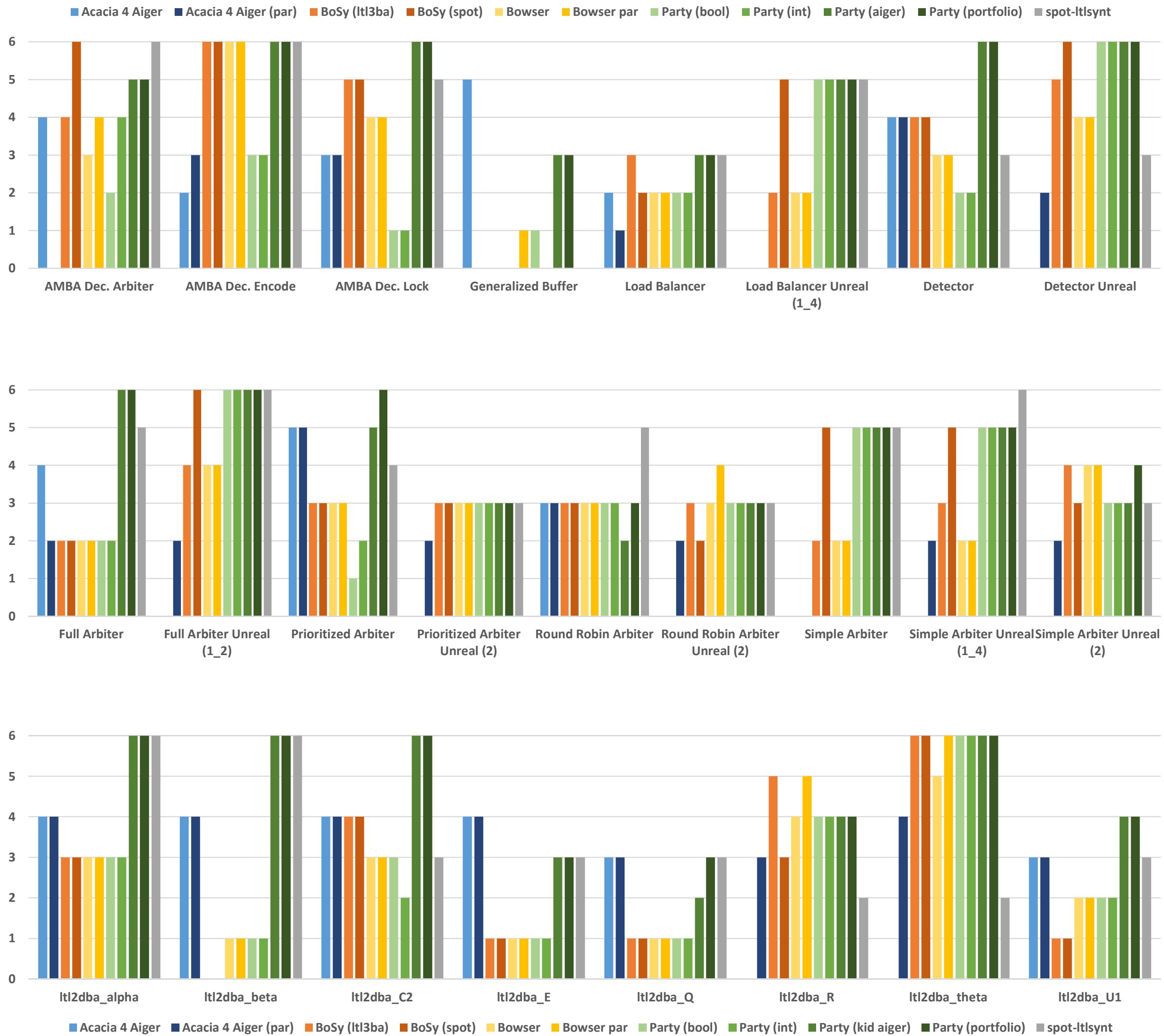

Figure 7: TLSF/LTL Realizability Track: Solved Instances for Parameterized Benchmarks

tions, PARTY with four configurations, and `ltlsynt` with one configuration. BoWSer competed in six configurations.

As for the AIGER/safety-track, there are two rankings in the synthesis subtrack, one based on the number of instances that can be solved, and the other based on the quality of solutions, measured by their size. Again, a solution for a realizable specification is only considered correct if it can be model-checked within a separate timeout of one hour (cf. Section 3). We start by presenting the results for the sequential configurations, followed by parallel configurations, and end with an analysis of the results.

Sequential Mode. Table 8 summarizes the experimental results for the sequential configurations, including the number of solved benchmarks, the uniquely solved instances, and the number of solutions that could not be model-checked within the timeout. The last column gives the accumulated quality points over all correct solutions.

As before, the "solved" column gives the number of problems that have either been correctly determined unrealizable, or for which the tool has presented a solution that could be verified. With this requirement, no sequential configuration could solve more than 200 or about 82% of the benchmarks, and 29 instances could not be solved by any tool. None of the tools provided any wrong solutions.

In this track, we note that all configurations that are not based on some form of bounded synthesis generated a significant number of solutions that could not be verified.

Table 8: Results: TLSF Synthesis (sequential mode only)

Tool	(configuration)	Solved	Unique	MC Timeout	Quality
Party	(aiger)	200	4	20	219
ltlsynt		182	3	13	180
BoSy	(spot)	181	3	0	**298**
Party	(int)	167	0	0	249
BoSy	(ltl3ba)	165	0	0	277
Party	(bool)	163	1	0	222
BoWSer	(c0)	162	0	0	273
BoWSer	(c1)	155	0	0	260
Acacia4Aiger		110	2	17	91
BoWSer	(c2)	93	0	0	141

Parallel Mode. In this mode, Acacia4Aiger, BoSy and BoWSer competed with parallel version of their configurations from the sequential track. Additionally, Party competed in a portfolio approach that combines its sequential configurations. Table 9 summarizes the experimental results, in the same format as before. No configuration solved more than 203 problem instances, or about 83% of the benchmark set. 27 benchmarks could not be solved by any tool. Party and Acacia4Aiger produced a significant number of solutions that could not be verified within the timeout. None of the solutions were determined to be wrong.

As before, we only consider instances as uniquely solved if they are not solved by any other configuration, including sequential ones. Consequently, none of the solutions are unique.

Table 9: Results: TLSF Synthesis (parallel mode only)

Tool	(configuration)	Solved	Unique	MC Timeout	Quality
Party	(portfolio)	203	0	18	**308**
BoSy	(spot,par)	181	0	0	297
BoSy	(ltl3ba,par)	169	0	0	286
BoWSer	(c0,par)	169	0	0	285
BoWSer	(c1,par)	169	0	0	285
BoWSer	(c2,par)	168	0	0	290
Acacia4Aiger	(par)	137	0	5	123

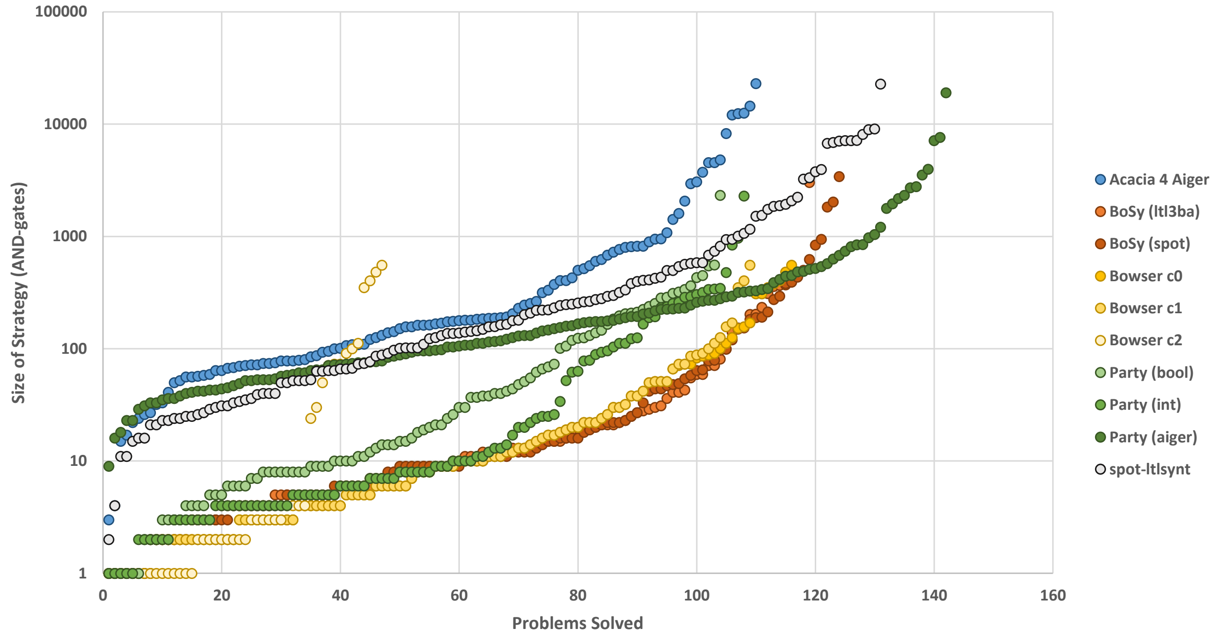

Figure 8: TLSF/LTL Synthesis Track: Solution Sizes of Sequential Configurations

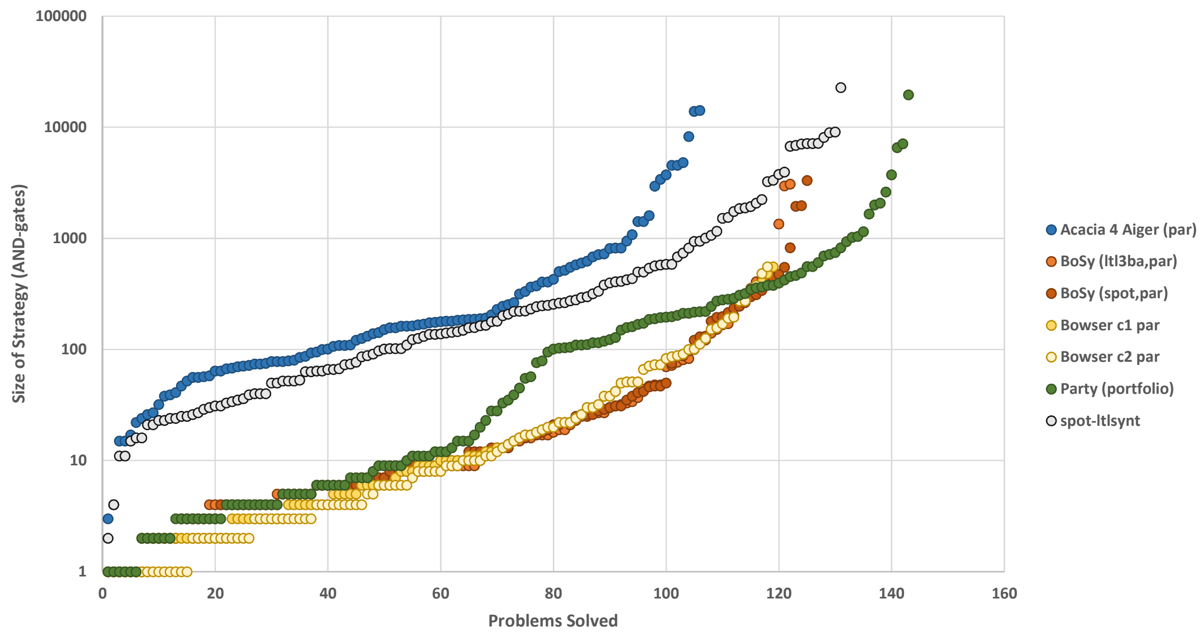

Figure 9: TLSF/LTL Synthesis Track: Solution Sizes of Parallel Configurations and `ltlsynt`

Analysis. As for the AIGER/safety-track, the number of solved instances for each tool in synthesis is closely related to the solved instances in realizability checking. The number of unique instances of Acacia4Aiger and PARTY (aiger) decreases, in part due to solutions that could not be verified. As a consequence, we also note that BoSy (spot) and PARTY (bool) now have some unique solutions, i.e., they do not provide the only solution, but the only solution that could be verified.

Considering the quality ranking, BoSy (spot) is the best tool in the sequential subtrack: even though it produces about 10% fewer solutions than PARTY (aiger), its accumulated quality points are about 36% higher. In fact, in the ranking based on quality, all bounded synthesis-based configurations are better than PARTY (aiger), except for BoWSer (c2). A different picture unfolds in the parallel subtrack, were PARTY (aiger) combines the strengths of its different approaches to find high-quality solutions for at least some of the benchmarks, such that it not only solves the highest number of problems, but also achieves the highest accumulated quality.

Figure 8 plots the sizes of sequential configurations. It shows, as expected, that the bounded synthesis approaches produce much smaller solutions than the other approaches. In particular, the solution sizes of all configurations of BoSy and BoWSer are very similar, with BoWSer producing more very small solutions (with < 10 AND-gates), and BoSy producing slightly better solutions for the remaining problems. The approach of PARTY (bool) falls somewhere in between. It also shows that the approach of PARTY (aiger) in many cases produces significantly smaller solutions than the approaches of Acacia4Aiger and `ltlsynt`.

In parallel mode, depicted in Figure 8, we can see changes mostly for BoWSer (c2,par) and PARTY (portfolio). The latter manages to combine the strengths of its different approaches, providing small solutions for those problems that can still be solved by its (int) configuration, and otherwise falling back to the (aiger) configuration. BoWSer (c2,par) shows the strength of its approach to produce the smallest possible solutions: for more than 60 benchmarks it provides the smallest solution.

A further analysis on the quality and the size of implementations shows that BoWSer (c2,par) is the configuration that has the highest average quality for the problems that it does solve. Furthermore, it produces the highest number of new reference solutions, i.e., solutions that have a quality greater than 2 according to the quality ranking scheme explained in Section 3.1. The analysis for all tools is given in Table 10.

6 Conclusions

SYNTCOMP 2017 consolidated the changes made last year, most importantly the introduction of the track for LTL specifications in the temporal logic synthesis format (TLSF). This year, two completely new tools have been entered in the competition: BoWSer and `ltlsynt`. Furthermore, four tools have received (sometimes major) updates, and four tools have been re-entered in the same version as last year. The only major change to the rules is the re-introduction of a quality ranking for the synthesis tracks.

In the AIGER/safety tracks, we had rather small changes on the tools and on the benchmark set, and this is reflected in similar, but not identical results as last year: Simple BDD Solver (abs1) again wins the sequential realizability mode, and the parallel realizability mode this year goes to TermiteSAT (hybrid), which was a close second to AbsSynthe last year. In the synthesis track, SafetySynth (basic) and AbsSynthe (PC1) again solve most problems in the sequential and parallel mode, respectively. In the quality ranking, SafetySynth (basic) also is the best configuration in sequential mode, and Demiurge (P3synt) is the best in parallel mode.

In the TLSF/LTL tracks, we had significant changes to both the tools and the benchmark set, including two new tools and a large number of new benchmarks. Consequently, the results look a lot different than last year. In fact, all of the winners are tools or configurations that did not participate last year: in sequential realizability and sequential synthesis, the new configuration (aiger) of PARTY solves most problems, followed by the new tool `ltlsynt`. In parallel realizability and synthesis, the new configuration PARTY (portfolio) solves most problems. Finally, in the quality ranking, new configuration BoSy

Table 10: TLSF Synthesis: Average Quality and New Reference Solutions

Tool	(configuration)	Avg. Quality	New Ref. Solutions
BoWSer	(c2,par)	1.725	50
BoSy	(ltl3ba,par)	1.691	31
BoWSer	(c1,par)	1.689	40
BoWSer	(c0,par)	1.688	40
BoWSer	(c0)	1.686	37
BoSy	(ltl3ba)	1.679	20
BoWSer	(c1)	1.676	34
BoSy	(spot)	1.644	30
BoSy	(spot,par)	1.643	23
PARTY	(portfolio)	1.517	27
BoWSer	(c2)	1.514	20
PARTY	(int)	1.493	20
PARTY	(bool)	1.363	15
PARTY	(aiger)	1.093	13
`ltlsynt`		0.988	8
Acacia4Aiger	(par)	0.898	0
Acacia4Aiger		0.825	0

(spot) has the highest accumulated quality in sequential mode, and PARTY (portfolio) wins the parallel mode.

Acknowledgments. The organization of SYNTCOMP 2016 was supported by the Austrian Science Fund (FWF) through project RiSE (S11406-N23) and by the German Research Foundation (DFG) through project "Automatic Synthesis of Distributed and Parameterized Systems" (JA 2357/2-1), and its setup and execution by the European Research Council (ERC) Grant OSARES (No. 683300).

The development of AbsSynthe and Acacia4Aiger was supported by an F.R.S.-FNRS and FWA fellowships, and the ERC inVEST (279499) project.

The development of SafetySynth and BoSy was supported by the ERC Grant OSARES (No. 683300).

References

[1] Luca de Alfaro & Pritam Roy (2010): *Solving games via three-valued abstraction refinement*. Inf. Comput. 208(6), pp. 666–676, doi:10.1016/j.ic.2009.05.007.

[2] Tomás Babiak, Mojmír Kretínský, Vojtech Rehák & Jan Strejcek (2012): *LTL to Büchi Automata Translation: Fast and More Deterministic*. In: *TACAS*, *LNCS* 7214, Springer, pp. 95–109, doi:10.1007/978-3-642-28756-5_8.

[3] Adrian Balint, Daniel Diepold, Daniel Gall, Simon Gerber, Gregor Kapler & Robert Retz (2011): *EDACC - An Advanced Platform for the Experiment Design, Administration and Analysis of Empirical Algorithms*. In: *LION 5. Selected Papers*, *LNCS* 6683, Springer, pp. 586–599, doi:10.1007/978-3-642-25566-3_46.

[4] R. Bloem, R. Könighofer & M. Seidl (2014): *SAT-Based Synthesis Methods for Safety Specs*. In: *VMCAI*, *LNCS* 8318, Springer, pp. 1–20, doi:10.1007/978-3-642-54013-4_1.

[5] Aaron Bohy, Véronique Bruyère, Emmanuel Filiot, Naiyong Jin & Jean-François Raskin (2012): *Acacia+, a Tool for LTL Synthesis*. In: *CAV*, *LNCS* 7358, Springer, pp. 652–657, doi:10.1007/978-3-642-31424-7_45.

[6] Robert K. Brayton & Alan Mishchenko (2010): *ABC: An Academic Industrial-Strength Verification Tool*. In: *CAV*, *LNCS* 6174, Springer, pp. 24–40, doi:10.1007/978-3-642-14295-6_5.

[7] Romain Brenguier, Guillermo A. Pérez, Jean-François Raskin & Ocan Sankur (2014): *AbsSynthe: abstract synthesis from succinct safety specifications*. In: *SYNT*, *EPTCS* 157, Open Publishing Association, pp. 100–116, doi:10.4204/EPTCS.157.11.

[8] Romain Brenguier, Guillermo A. Pérez, Jean-François Raskin & Ocan Sankur (2015): *Compositional Algorithms for Succinct Safety Games*. In: *SYNT*, *EPTCS* 202, Open Publishing Association, pp. 98–111, doi:10.4204/EPTCS.202.7.

[9] Gianpiero Cabodi, Carmelo Loiacono, Marco Palena, Paolo Pasini, Denis Patti, Stefano Quer, Danilo Vendraminetto, Armin Biere & Keijo Heljanko (2016): *Hardware Model Checking Competition 2014: An Analysis and Comparison of Solvers and Benchmarks*. *Journal on Satisfiability, Boolean Modeling and Computation* 9, pp. 135–172. Available at `https://www.satassociation.org/jsat/index.php/jsat/article/download/129/111`.

[10] Cristian S. Calude, Sanjay Jain, Bakhadyr Khoussainov, Wei Li & Frank Stephan (2017): *Deciding parity games in quasipolynomial time*. In Hamed Hatami, Pierre McKenzie & Valerie King, editors: *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, ACM, pp. 252–263, doi:10.1145/3055399.3055409.

[11] Alonzo Church (1962): *Logic, arithmetic and automata*. In: *Proceedings of the international congress of mathematicians*, pp. 23–35.

[12] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault & Laurent Xu (2016): *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)*, *LNCS* 9938, Springer, pp. 122–129, doi:10.1007/978-3-319-46520-3_8.

[13] Rüdiger Ehlers (2012): *Symbolic bounded synthesis*. *Formal Methods in System Design* 40(2), pp. 232–262, doi:10.1007/s10703-011-0137-x.

[14] Peter Faymonville, Bernd Finkbeiner, Markus N. Rabe & Leander Tentrup (2017): *Encodings of Bounded Synthesis*. In: *TACAS (1)*, *LNCS* 10205, pp. 354–370, doi:10.1007/978-3-662-54577-5_20.

[15] Peter Faymonville, Bernd Finkbeiner & Leander Tentrup (2017): *BoSy: An Experimentation Framework for Bounded Synthesis*. In: *CAV (2)*, *LNCS* 10427, Springer, pp. 325–332, doi:10.1007/978-3-319-63390-9_17.

[16] Bernd Finkbeiner & Felix Klein (2016): *Bounded Cycle Synthesis*. In: *CAV (1)*, *LNCS* 9779, Springer, pp. 118–135, doi:10.1007/978-3-319-41528-4_7.

[17] Bernd Finkbeiner, Markus N. Rabe & César Sánchez (2015): *Algorithms for Model Checking HyperLTL and HyperCTL**. In: *CAV*, *LNCS* 9206, Springer, pp. 30–48, doi:10.1007/978-3-319-21690-4_3.

[18] Bernd Finkbeiner & Sven Schewe (2013): *Bounded synthesis*. *STTT* 15(5-6), pp. 519–539, doi:10.1007/s10009-012-0228-z.

[19] Johann Glaser & Clifford Wolf (2014): *Methodology and Example-Driven Interconnect Synthesis for Designing Heterogeneous Coarse-Grain Reconfigurable Architectures*, pp. 201–221. Springer International Publishing, Cham, doi:10.1007/978-3-319-01418-0_12.

[20] Swen Jacobs (2014): *Extended AIGER Format for Synthesis*. *CoRR* abs/1405.5793. Available at `http://arxiv.org/abs/1405.5793`.

[21] Swen Jacobs & Roderick Bloem (2016): *The Reactive Synthesis Competition: SYNTCOMP 2016 and Beyond*. In: *SYNT@CAV 2016*, *EPTCS* 229, pp. 133–148, doi:10.4204/EPTCS.229.11.

[22] Swen Jacobs, Roderick Bloem, Romain Brenguier, Rüdiger Ehlers, Timotheus Hell, Robert Könighofer, Guillermo A. Pérez, Jean-François Raskin, Leonid Ryzhyk, Ocan Sankur, Martina Seidl, Leander Tentrup

& Adam Walker (2017): *The first reactive synthesis competition (SYNTCOMP 2014).* STTT 19(3), pp. 367–390, doi:10.1007/s10009-016-0416-3.

[23] Swen Jacobs, Roderick Bloem, Romain Brenguier, Ayrat Khalimov, Felix Klein, Robert Könighofer, Jens Kreber, Alexander Legg, Nina Narodytska, Guillermo A. Pérez, Jean-François Raskin, Leonid Ryzhyk, Ocan Sankur, Martina Seidl, Leander Tentrup & Adam Walker (2016): *The 3rd Reactive Synthesis Competition (SYNTCOMP 2016): Benchmarks, Participants & Results.* In: SYNT@CAV, EPTCS 229, pp. 149–177, doi:10.4204/EPTCS.229.12.

[24] Swen Jacobs, Roderick Bloem, Romain Brenguier, Robert Könighofer, Guillermo A. Pérez, Jean-François Raskin, Leonid Ryzhyk, Ocan Sankur, Martina Seidl, Leander Tentrup & Adam Walker (2016): *The Second Reactive Synthesis Competition (SYNTCOMP 2015).* In: SYNT, EPTCS 202, Open Publishing Association, pp. 27–57, doi:10.4204/EPTCS.202.4.

[25] Swen Jacobs, Felix Klein & Sebastian Schirmer (2016): *A High-Level LTL Synthesis Format: TLSF v1.1.* In: SYNT@CAV 2016, EPTCS 229, pp. 112–132, doi:10.4204/EPTCS.229.10.

[26] Mikolás Janota, William Klieber, Joao Marques-Silva & Edmund M. Clarke (2016): *Solving QBF with counterexample guided refinement.* Artif. Intell. 234, pp. 1–25, doi:10.1016/j.artint.2016.01.004.

[27] Barbara Jobstmann (2007): *Applications and Optimizations for LTL Synthesis.* Ph.D. thesis, Graz University of Technology.

[28] Barbara Jobstmann & Roderick Bloem (2006): *Optimizations for LTL Synthesis.* In: FMCAD, IEEE Computer Society, pp. 117–124, doi:10.1109/FMCAD.2006.22.

[29] Ayrat Khalimov (2015): *Specification Format for Reactive Synthesis Problems.* In: SYNT, EPTCS 202, Open Publishing Association, pp. 112–119, doi:10.4204/EPTCS.202.8.

[30] Ayrat Khalimov, Swen Jacobs & Roderick Bloem (2013): *PARTY Parameterized Synthesis of Token Rings.* In: CAV, LNCS 8044, Springer, pp. 928–933, doi:10.1007/978-3-642-39799-8_66.

[31] Ayrat Khalimov, Swen Jacobs & Roderick Bloem (2013): *Towards Efficient Parameterized Synthesis.* In: VMCAI, LNCS 7737, Springer, pp. 108–127, doi:10.1007/978-3-642-35873-9_9.

[32] Alexander Legg, Nina Narodytska & Leonid Ryzhyk (2016): *A SAT-Based Counterexample Guided Method for Unbounded Synthesis.* In: CAV (2), LNCS 9780, Springer, pp. 364–382, doi:10.1007/978-3-319-41540-6_20.

[33] Jia Hui Liang, Vijay Ganesh, Pascal Poupart & Krzysztof Czarnecki (2016): *Learning Rate Based Branching Heuristic for SAT Solvers.* In Nadia Creignou & Daniel Le Berre, editors: Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings, Lecture Notes in Computer Science 9710, Springer, pp. 123–140, doi:10.1007/978-3-319-40970-2_9.

[34] Donald A. Martin (1975): *Borel Determinacy.* Annals of Mathematics 102(2), pp. 363–371.

[35] Leonardo Mendonça de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver.* In: TACAS, LNCS 4963, Springer, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[36] Roman R. Redziejowski (2012): *An Improved Construction of Deterministic Omega-automaton Using Derivatives.* Fundam. Inform. 119(3-4), pp. 393–406, doi:10.3233/FI-2012-744.

[37] Olivier Roussel (2011): *Controlling a Solver Execution with the runsolver Tool.* JSAT 7(4), pp. 139–144. Available at `http://jsat.ewi.tudelft.nl/content/volume7/JSAT7_12_Roussel.pdf`.

[38] Leonid Ryzhyk, Adam Walker, John Keys, Alexander Legg, Arun Raghunath, Michael Stumm & Mona Vij (2014): *User-Guided Device Driver Synthesis.* In: OSDI, USENIX Association, pp. 661–676. Available at `https://www.usenix.org/conference/osdi14/technical-sessions/presentation/ryzhyk`.

[39] M. Seidl & R. Könighofer (2014): *Partial witnesses from preprocessed quantified Boolean formulas.* In: DATE'14, IEEE, pp. 1–6, doi:10.7873/DATE2014.162.

[40] Leander Tentrup (2016): *Solving QBF by Abstraction.* CoRR abs/1604.06752. Available at `http://arxiv.org/abs/1604.06752`.

[41] Cong Tian, Jun Song, Zhenhua Duan & Zhao Duan (2015): *LtlNfBa: Making LTL Translation More Practical*. In: *SOFL+MSVL*, *LNCS* 9559, Springer, pp. 179–194, doi:10.1007/978-3-319-31220-0_13.

[42] James C. Tiernan (1970): *An efficient search algorithm to find the elementary circuits of a graph*. *Commun. ACM* 13(12), pp. 722–726, doi:10.1145/362814.362819.

[43] Cheng-Yin Wu, Chi-An Wu, Chien-Yu Lai & Chung-Yang R. Haung (2014): *A Counterexample-Guided Interpolant Generation Algorithm for SAT-Based Model Checking*. *IEEE Trans. on CAD of Integrated Circuits and Systems* 33(12), pp. 1846–1858, doi:10.1109/TCAD.2014.2363395.

[44] Wieslaw Zielonka (1998): *Infinite Games on Finitely Coloured Graphs with Applications to Automata on Infinite Trees*. *Theor. Comput. Sci.* 200(1-2), pp. 135–183, doi:10.1016/S0304-3975(98)00009-7.