The First Reactive Synthesis Competition (SYNTCOMP 2014)

Swen Jacobs[1,2], Roderick Bloem[1], Romain Brenguier[3], Rüdiger Ehlers[4,5], Timotheus Hell[1], Robert Könighofer[1], Guillermo A. Pérez[3], Jean-François Raskin[3], Leonid Ryzhyk[6,7], Ocan Sankur[8,3], Martina Seidl[9], Leander Tentrup[2], Adam Walker[6]

[1] Graz University of Technology, Austria
[2] Saarland University, Saarbrücken, Germany
[3] Université Libre de Bruxelles
[4] University of Bremen, Germany
[5] DFKI GmbH, Bremen, Germany
[6] NICTA, Sydney, Australia
[7] Carnegie Mellon University, Pittsburgh, USA
[8] CNRS, IRISA, Rennes, France
[9] Johannes-Kepler-University Linz, Austria

Abstract. We introduce the reactive synthesis competition (SYNTCOMP), a long-term effort intended to stimulate and guide advances in the design and application of synthesis procedures for reactive systems. The first iteration of SYNTCOMP is based on the controller synthesis problem for finite-state systems and safety specifications. We provide an overview of this problem and existing approaches to solve it, and report on the design and results of the first SYNTCOMP. This includes the definition of the benchmark format, the collection of benchmarks, the rules of the competition, and the five synthesis tools that participated. We present and analyze the results of the competition and draw conclusions on the state of the art. Finally, we give an outlook on future directions of SYNTCOMP.

Key words: synthesis, reactive systems, competition, experimental evaluation, benchmarks, safety games

1 Introduction

Ever since its definition by Church [23], the automatic synthesis of reactive systems from formal specifications has been one of the major challenges of computer science, and an active field of research. A number of fundamental approaches to solve the problem have been proposed (see e.g. [31, 53, 54]). Despite the obvious advantages of automatic synthesis over manual implementation and the significant progress of research on theoretical aspects of synthesis, the impact of formal synthesis procedures in practice has been very limited. One reason for this limited impact is the scalability problem that is inherent to synthesis approaches. The reactive synthesis problem is in general 2EXPTIME-complete for LTL specifications [53]. A number of approaches have recently been invented to solve special cases of the problem more efficiently, either by restricting the specification language [12], or by a smart exploration of the search space [29, 32–35, 59]. While important progress on the scalability problem has been made, an additional problem is the lacking maturity and comparability of implementations, and a lack of incentive for the development of efficient implementations [27]. Solving different aspects of this problem is the main motivation of SYNTCOMP, as explained in the following (inspired by [48]).

Synthesis tools are hard to compare. Research papers that introduce a new algorithm in many cases do include a comparison of its implementation against existing ones. However, the comparison of a large number of tools on a benchmark set of significant size can take weeks or months of computation time. This is often circumvented in research papers by comparing the new results to existing experimental data (usually obtained under different experimental conditions), or by comparing against a small number of tools on a small benchmark set. In both cases, this limits the value of the experimental results. In contrast, SYNTCOMP provides reliable results for a significant number of synthesis tools on a large benchmark set, with consistent experimental conditions.

It is hard to exchange benchmark sets. Related to the comparison of tools, we note that almost every existing tool uses its own input language, and benchmarks have to be translated from one format to another in order to compare different tools. This makes it hard to exchange benchmark sets, and adds another source of uncertainty when comparing tools. SYNTCOMP aims to solve these issues by defining a standard benchmark format, and by

collecting a benchmark library that is publicly available for the research community.

Usability of synthesis tools. Implementations of many synthesis approaches do exist [8, 14, 28], but they cannot effectively be used as *black-box solvers for applications*. The definition of a standard language is a first step in this direction. In addition, the competition forces tool developers to produce implementations that are sufficiently robust to work on the complete benchmark library of SYNTCOMP with a fixed configuration. Thus, SYNTCOMP promotes the *simplicity of use* that comes with push-button approaches that do not require any user intervention.

Summing up, the goal of the *reactive synthesis competition* (SYNTCOMP) is to foster research in scalable and user-friendly implementations of synthesis techniques.

Related competitions. Competitions have been used to achieve these goals in many related fields, including automated reasoning [5, 43, 62] and automated verification [6][1]. A difference of synthesis competitions to most of the competitions in automated reasoning or verification is that solutions to the synthesis problem can be ranked according to inherent quality criterions that go beyond mere correctness, such as reaction time or size of the solution. Thus, a synthesis competition also needs to measure the quality of solutions with respect to these additional metrics.

In parallel to SYNTCOMP 2014, the *syntax-guided synthesis competition* (SyGuS-COMP) was held for the first time [1]. The focus of SyGuS-COMP is on the synthesis of functional instead of reactive programs, and the specification is given as a first-order logic constraint on the function to be synthesized, along with a syntactic constraint that restricts how solutions can be built. The goals of SyGuS-COMP are similar to those of SYNT-COMP, but for a fundamentally different class of programs and specifications.

Timeline. The organization of the first SYNTCOMP began formally with a presentation and discussion of ideas at the second Workshop on Synthesis (SYNT) in July 2013. The organization team consisted of Roderick Bloem, Rüdiger Ehlers and Swen Jacobs. The decision for the specification format was made and announced in August 2013, and a call for benchmarks, along with the rules of the competition, was published in November 2013. In March 2014 we published our reference implementation, and benchmarks were collected until the end of April 2014. Participants had to submit their tools until the end of May 2014, and the experiments for the competition were executed in June and July 2014. The results were first presented at the 26th International Conference on Computer Aided Verification (CAV) and the 3rd SYNT Workshop in July 2014.

---

[1] See also: the Hardware Model Checking Competition, `http://fmv.jku.at/hwmcc/`. Accessed February 2016.

Goals. The first competition had the following goals:

- define a class of synthesis problems and a benchmark format that results in a low entry-barrier for interested researchers to enter the competition
- collect benchmarks in the SYNTCOMP format
- encourage development of synthesis tools that support the SYNTCOMP format
- provide a lobby that connects tool developers with each other, and with possible users of synthesis tools

SYNTCOMP 2014 was already a success before the experimental evaluation began: within less than 10 months after the definition of the benchmark format, we collected 569 benchmark instances in 6 classes of benchmarks, and 5 synthesis tools from 5 different research groups were entered into the competition. For four of the tools, at least one of the developers was present at CAV and/or the SYNT workshop.

Overview. The rest of this article describes the background, design, participating solvers (called *entrants*), and results of SYNTCOMP 2014. We will introduce the synthesis problem for safety specifications, as well as different approaches for solving it, in Section 2. We define the SYNTCOMP format in Section 3, and describe the benchmark set for SYNTCOMP 2014 in Section 4. Section 5 defines the rules of the competition. In Section 6 we give an overview of the entrants of SYNTCOMP 2014, followed by some notes on the execution of the competition in Section 7. In Section 8, we present and analyze the experimental results of the competition.

Note that Sections 4 and 6, as well as parts of Section 2 are based on the descriptions that the respective benchmark and tool authors had to supply in order to participate. The setup of the competition framework as described in Section 7 was taken care of by Timotheus Hell. The remainder of this article is original work of the SYNTCOMP organizers.

This article is based on the first description of the SYNTCOMP format [39] and a preliminary version of the SYNTCOMP 2014 report [40].

2 Problem Description and Synthesis Approaches

Informally, the reactive synthesis problem consists of finding a system C that satisfies a given specification φ in an adversarial environment. In general, systems may be infinite-state (programs) or finite-state (circuits), and specifications can come in different forms, for example as temporal logic formulas or as monitor circuits.

For the first SYNTCOMP, we aimed for a low entry-barrier for participants, and to keep the competition manageable in terms of tasks like the definition of input and output format, and the verification of results. To this end, we only consider the synthesis of finite-state

systems from pure safety specifications modeled as monitor circuits. The monitor circuit reads two kinds of input signals: uncontrollable inputs from the environment, and controllable inputs from the system to be synthesized. It raises a special output BAD if the safety property φ has been violated by the sequence of input signal valuations it has read thus far.

Then, the *realizability problem* is to determine if there exists a circuit C that reads valuations of the uncontrollable inputs and provides valuations of the controllable inputs such that BAD is not raised on any possible execution. The *synthesis problem* is to provide such a C if it exists. As a quality criterion, we consider the *size* of the produced implementation, which not only correlates to the cost of implementing a circuit in hardware, but often also leads to implementations which have other desirable properties, like short reaction time.

2.1 Synthesis as a Safety Game

The traditional approach to reactive synthesis is to view the problem as a game between two players [20, 54, 63]: the *environment player* decides uncontrollable inputs, and the *system player* decides controllable inputs of the monitor circuit. *States* of the game are valuations of the latches in the monitor circuit. A state is *safe* if BAD is not raised. The goal of the system player is to visit only safe states, regardless of the environment behavior.

Game-based Synthesis. In a first step, a so-called *winning region* for the system player is computed. The winning region W is the set of all states from which the system player can enforce the specification, i.e., from which it can guarantee that the environment cannot force the game into an unsafe state.

In a second step, a *winning strategy* is derived from the winning region. For every state and every valuation of the uncontrollable inputs, the winning strategy defines a set of possible valuations of the controllable inputs that can ensure that the winning region is not left.

The last step is to implement this strategy in a circuit, where a concrete choice for the controllable inputs has to be made for every state and valuation of uncontrollable inputs.

All of the tools in SYNTCOMP 2014 implement such a game-based synthesis approach, in one form or another.

Symbolic encoding. To achieve acceptable scalability, it is important to implement synthesis algorithms symbolically, i.e., by manipulating formulas instead of enumerating states [2]. In synthesis, symbolic algorithms are usually implemented with Binary Decision Diagrams (BDDs) [19, 60]. Most of the tools in SYNTCOMP 2014 use BDD-based approaches with different optimizations to achieve good performance in synthesis time and circuit size.

However, BDDs also have scalability issues, in particular the growing size of the data structure itself. Alternatively, the problem can be encoded into a sequence of propositional satisfiability (SAT), quantified Boolean formulas (QBF), or satisfiability modulo theories (SMT) problems. The enormous performance improvements in decision procedures for satisfiability over the last decades encourage such approaches.

In the following, we give a mostly informal description of the three synthesis techniques used by the tools that entered SYNTCOMP 2014: BDD-based game solving (Section 2.3), SAT-/QBF-based game solving (Section 2.4), and template-based synthesis (Section 2.5).

2.2 Preliminaries: Circuits and Games

Let $\mathbb{B} = \{0, 1\}$. If X denotes a finite set of Boolean variables, then any $v \in \mathbb{B}^X$ is called a *valuation of X*. Sets of valuations of X are represented by *quantified Boolean formulas* on X, which are made of propositional logic and first-order quantification on X. A *formula f* with free variables X will be written as $f(X)$, and for the same formula under a given valuation v of X we write $f(v)$. If the free variables are $X \cup Y$, we also write $f(X, Y)$. For a set of variables $X = \{x_1, \ldots, x_n\}$, we write $\exists X$ instead of $\exists x_1 \exists x_2 \ldots \exists x_n$, and similarly for universal quantification. For a set of variables $X = \{x_1, \ldots, x_n\}$, we use X' to denote $\{x'_1, \ldots, x'_n\}$, a set of primed copies of the variables in X, usually representing the variables after a step of the transition relation.

Then, the synthesis problem is given as a (sequential) monitor circuit M over the sets of variables L, X_u, X_c, where

- L are state variables for the latches in the monitor circuit,
- X_u are uncontrollable input variables,
- X_c are controllable input variables, and
- BAD $\in L$ is a special variable for the unsafe states, i.e., a state is unsafe iff BAD $= 1$.[2]

We assume that the system has a unique initial state, in which all latches in L including BAD are initialized to 0.

A *solution* of the synthesis problem is a sequential circuit C with inputs $L \cup X_u$ and outputs X_c, such that the composition of C and M is safe, i.e., states with BAD $= 1$ are never reached, for any sequence of (uncontrollable) inputs X_u and starting from the unique initial state. The synthesis problem is depicted in Figure 1.

Note that a circuit defines a (Mealy-type) *finite-state machine* in the standard way. With the additional distinction between controllable and uncontrollable inputs and the interpretation of BAD as the set of unsafe states,

---

[2] For simplicity, we assume that BAD is a latch. If BAD is not a latch in the given monitor circuit, then it can be described as a formula $f(L, X_u, X_c)$. In this case we can obtain a problem in the described form by extending the circuit with a new latch that takes $f(L, X_u, X_c)$ as an input and provides output BAD.
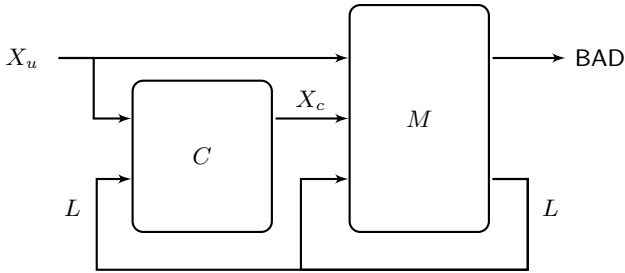
Fig. 1. Synthesis problem with monitor circuit M and (unknown) system circuit C

it defines a *safety game*: the set of states is $\mathbb{B}^L$ (valuations of latches L), with initial state 0^L. The transition relation of the monitor circuit can be translated into a formula $T(L, X_u, X_c, L')$ that relates valuations of L, X_u, X_c to the valuation of (next- state variables) L'. In every turn of the game, first the environment player chooses a valuation of X_u, and then the system player chooses a valuation of X_c. The successor state is computed according to $T(L, X_u, X_c, L')$. A *strategy* of the system player is a function that maps the sequence of valuations of L and X_u seen thus far to a set of possible valuations for X_c. It is *deterministic* if it always maps to a unique valuation. A strategy is *winning* for the system player if it avoids entering the unsafe states regardless of the actions of the environment. Two-player safety games are *determined*, i.e., for every such game either the environment player or the system player has a winning strategy. A *memoryless* strategy only depends on the current values of L and X_u. For safety games, there exists a winning strategy iff there exists a memoryless winning strategy. A deterministic memoryless winning strategy can be represented as a circuit, and thus provides a solution to the synthesis problem.

2.3 BDD-based Game Solving

For a basic BDD-based algorithm, assume that the transition relation $T(L, X_u, X_c, L')$ and the sets of initial and unsafe states are each represented as a single BDD (see e.g. [24]). To determine whether the environment has a strategy that guarantees its victory, one repeatedly computes the set of states from which it can force the game into the unsafe states. If $S(L')$ is a formula over the latches L', representing a set of states, then the set of *uncontrollable predecessors* of $S(L')$ can be computed as the set of valuations of latches L that satisfy

$$\mathsf{UPRE}(S(L')) = \exists X_u \,\forall X_c \,\exists L'. \; S(L') \wedge T(L, X_u, X_c, L').$$

To compute the winning region $W(L)$ of the system player, we first compute the least fixpoint of UPRE on BAD:

$$\mu S(L). \; \mathsf{UPRE}(S(L') \vee \mathsf{BAD}') \qquad (1)$$

The resulting set of latch valuations represents the states from which the environment can force the game into the unsafe states. Since two-player safety games are determined, the complement of this set is the winning region $W(L)$ for the system player (see, e.g. [63]).

That is, if during our fixpoint computation we notice that the environment can force the game from the initial state to the unsafe states, then we can stop — the specification is unrealizable. Otherwise, the initial state will be contained in the winning region $W(L)$, and $W(L)$ represents a non-deterministic strategy λ for the system player, which can be described as a function λ that maps a valuation $s \in \mathbb{B}^L$ of the latches and a valuation $\sigma_u \in \mathbb{B}^{X_u}$ of the uncontrollable variables to a set of possible valuations for the controllable variables:

$$\lambda(s, \sigma_u) = \{\sigma_c \in \mathbb{B}^{X_c} \mid \forall L'. \; T(s, \sigma_u, \sigma_c, L') \to W(L')\}.$$

To solve the synthesis problem, in principle any determinization of λ can be chosen to obtain a functional strategy for the system player.

In order to compute the winning region efficiently and to find a strategy that can be represented as a small circuit, a number of optimizations can be used. We introduce some of the common optimizations in the following, in order to be able to compare and distinguish the participants that use a BDD-based approach.

Partitioned Transition Relation and Direct Substitution. To be efficient, the explicit construction of the BDD for the transition relation should be avoided. This can be achieved by *partitioning* the transition relation into a set of simpler relations, inspired by similar approaches for the model checking problem [21,24,55]. A common approach is to split $T(L, X_u, X_c, L')$ into a set of (functional) relations $\{f_l(L, X_u, X_c)\}_{l \in L}$, where each f_l represents the next-state value of latch l.

Then, the uncontrollable predecessor can be computed as

$$\mathsf{UPRE}(S(L')) = \exists X_u \,\forall X_c. \; S(L')[l' \leftarrow f_l(L, X_u, X_c)]_{l \in L},$$

avoiding to ever build the monolithic transition relation, as well as having to ever declare a next state copy of any latch variable in the BDD manager. Substituting individual latches with functional BDDs is directly supported by existing BDD packages, e.g., function BDDVECTOR-COMPOSE in the CUDD package [60]. This will be called *partitioned transition relation* in the tool descriptions.

A special case of this approach is to identify those latches that only store the value of some other variable from the last step, i.e., the latch update function has the form $f_l = x$ for some $x \in L \cup X_u \cup X_c$, and use the substitution of latches by f_l only for them. In this case, we only substitute with a single existing variable instead of a functional BDD, which can be done, e.g., with CUDD's BDDVARMAP method. This will be called *direct substitution* in the tool descriptions.

BDD Reordering. Efficiency of BDD-based algorithms depends to a large extent on the size of BDDs, which in turn depends on the variable ordering [3]. To keep the data structures small, *reordering* heuristics are commonly used to try to minimize BDDs at runtime [57]. Standard BDD packages come with pre-defined reordering strategies. Algorithms that do not use reordering at all are usually not competitive.

Efficient Computation of UPRE. In the fixpoint computation, we repeatedly use the UPRE operation to compute the set from which the environment wins the game. This operation consists of conjoining the current set of states with the transition relation, followed by resolving the quantification over inputs and current states to get the description of the set of predecessor states. The latter is called (existential or universal) *abstraction* over these variables. In practice, it is often preferable to not use this strict order, but instead do conjunction and abstraction in parallel. This is directly supported in some BDD packages, e.g., in CUDD's BDDANDABSTRACT method. We will call this optimization *simultaneous conjunction and abstraction*.

Eager Deallocation of BDDs. Another optimization is to deallocate BDDs that are no longer needed as soon as possible. Not only do these BDDs take up memory, but more importantly they are also counted and analyzed for BDD reordering. Thus, removing such BDDs saves space and time. We will call this *eager deallocation*.

Abstraction-based algorithms. For systems that have a large state space, an *abstraction approach* may be more efficient than precise treatment of the full state space [25, 36, 37]. This can be done by defining a set of predicates P (which may simply be a subset of the state variables of the system [26, 47]) and computing over- and under-approximations of the UPRE function, with respect to the partitioning of the state space defined by the predicates in P. Computing fixpoints for these approximations is usually much cheaper than computing the precise fixpoint for the system. If the system player wins the game for the over-approximation of UPRE, then it also wins the original game. If the system player loses for the under-approximation of UPRE, then it also loses the original game. If neither is the case, then the abstraction is insufficient and needs to be refined by introducing additional predicates.

Extraction of Small Winning Strategies. To obtain from λ a functional strategy that can be represented as a small circuit, a number of optimizations is commonly used. To this end, let λ_c be the restriction of λ to one output $c \in X_c$. We want to obtain a partitioned strategy, represented as one function $f_c(L, X_u)$ for every $c \in X_c$:

- For every $c \in X_c$, in some arbitrary order, compute the positive and negative co-factors of λ_c, i.e., the values s, σ_u for which $\lambda_c(s, \sigma_u)$ can be 1 or 0, respectively. These can be used to uniquely define f_c, e.g.,

by letting $f_c(s, \sigma_u) = 0$ for all values in the negative co-factor, and $f_c(s, \sigma_u) = 1$ otherwise [11]. This will be called *co-factor-based extraction of winning strategies*.

- After extracting the functions $f_c(L, X_u)$ for all $c \in X_c$, one can minimize the strategy by doing an additional *forward reachability analysis*: compute the reachable states *with this strategy*, and restricting all f_c to values of L that are actually reachable.

- After translating the functions $f_c(L, X_u)$ into an AIG representation, a number of minimization techniques can be used to obtain small AIGs [49, 50]. The verification tool ABC[3] [17] implements a number of these minimization strategies that can be used in a black-box fashion to obtain smaller circuits, and we will call this approach *ABC minimization*.

2.4 Incremental SAT- and QBF-based Game Solving

In contrast to the BDD-based approaches already presented, the SAT- and QBF-based approaches start with a coarse over-approximation of the winning region, represented as a CNF formula $W(L)$ over the state variables L. This approximation is incrementally refined, such that $W(L)$ eventually represents the winning region symbolically.

More concretely, we initialize $W(L)$ to the set of all safe states ¬BAD. In each iteration, the underlying solver is used to compute a state $s \models W(L) \wedge \mathsf{UPRE}(\neg W(L'))$ within the current candidate version $W(L)$ of the winning region from which the environment player can enforce to leave $W(L)$ in one step. Obviously, such a state cannot be part of the winning region. Hence, we refine $W(L)$ by removing this state. The state s can be represented as a cube over the state variables L, so removing s from $W(L)$ amounts to adding the clause $\neg s$.

In order to remove a larger region from $W(L)$, the algorithm tries to generalize the clause $\neg s$ by removing literals, as long as the resulting clause $\neg \tilde{s}$ still only excludes states from which $\neg W(L)$ can be reached by the environment in one step. More specifically, literals are dropped as long as $(W(L) \wedge \neg \tilde{s}) \rightarrow \mathsf{UPRE}(\neg W(L'))$ holds. Once $W(L) \wedge \mathsf{UPRE}(\neg W(L'))$ becomes unsatisfiable, i.e., no more state exists from which the environment can enforce to leave $W(L)$, we have found the final winning region and the algorithm terminates.

Implementation and Optimizations. A simple realization of this approach uses a QBF solver both to compute a state s and to generalize the induced blocking clause $\neg s$. A generally more efficient approach is to use two competing SAT solvers for the two different quantifiers in UPRE when computing s. Other optimizations

---

[3] http://www.eecs.berkeley.edu/~alanmi/abc/. Accessed February 2016.

include the utilization of reachability information during the computation of s and during the generalization of $\neg s$. A detailed description of different realizations and optimizations can be found in [13].

Extraction of Small Winning Strategies. To obtain an implementation from the winning region, different methods can be applied. One possibility is to compute a certificate for the validity of the QBF formula

$$\forall L, X_u \; \exists X_c, L'. \; W(L) \to \big(T(L, X_u, X_c, L') \wedge W(L')\big)$$

in the form of functions defining the variables in X_c based on L and X_u using methods for QBF certification [52]. Another option are learning-based approaches that have also been proposed for BDD-based synthesis, but work particularly well in a SAT-/QBF-based framework [9, 30]. Similar to the BDD-based methods for extracting small winning strategies, these learning approaches also compute solutions for one output $c \in X_c$ at a time. They start with a first guess of a concrete output function, and then refine this guess based on counterexamples.

2.5 Template-based Synthesis

In order to compute a winning region, symbolically represented as a formula $W(L)$ over the state variables L, this approach constructs a parameterized CNF formula $\tilde{W}(L, P)$, where P is a certain set of Boolean template parameters. Different concrete values for these parameters P induce a different concrete CNF formula $W(L)$ over the state variables. This is done as follows. First, the approach fixes a maximum number N of clauses. Then, for every clause and every state variable, it introduces parameters defining whether the state variable occurs in the clause, whether it occurs negated or unnegated, and whether the clause is used at all. This way, the search for a CNF formula over the state variables (the winning region $W(L)$) is reduced to a search for Boolean constants (values for the template parameters P). A QBF solver is used to compute template parameter values such that (a) the winning region contains only safe states, (b) the winning region contains the initial state, and (c) from each state of the winning region, the system player can guarantee that the successor state will also be in the winning region, regardless of the choice of the environment. This is done by computing a satisfying assignment for the variables P in QBF:

$$\exists P \; \forall L, X_u \; \exists X_c, L'.$$
$$\tilde{W}(L, P) \to \neg \mathsf{BAD} \; \wedge$$
$$\mathrm{Init}(L) \to \tilde{W}(L, P) \; \wedge$$
$$\tilde{W}(L, P) \to \big(T(L, X_u, X_c, L') \wedge \tilde{W}(L', P)\big).$$

More details can be found in [13].

3 Benchmark Format

For the first SYNTCOMP, we have chosen to use an extension of the AIGER format that is already used in automatic verification and is suitable for our selected range of problems, as well as extendable to other classes of problems. Furthermore, the format poses a low entry-barrier for developers of synthesis tools, as synthesis problems are directly given in a bit-level representation that can easily be encoded into BDDs and SAT-based approaches. In the following, we first recapitulate the AIGER format[4], defined by Biere as the specification language for the hardware model checking competition (HWMCC)[5]. Then we show an extension of AIGER to a specification format for synthesis problems with safety specifications, developed for SYNTCOMP. Finally, we define how to use the AIGER format for solutions of synthesis problems in this setting.

3.1 Original AIGER Format

The AIGER format was developed as a compact and simple file format to represent benchmarks for the hardware model checking competition (HWMCC). Benchmarks are encoded as multi-rooted And-Inverter Graphs (AIGs) with latches that store the system state. We use version `20071012` of the format. There is an ASCII variant and a more compact binary variant of the format. Since the binary format is more restricted and thus harder to extend than the ASCII format, we have chosen to work with the ASCII variant for SYNTCOMP. In the following, we explain the structure of AIGER files for model checking of safety properties.

A file in AIGER format (ASCII variant) consists of the following parts:

1. Header,
2. Input definitions,
3. Latch definitions,
4. Output definitions,
5. AND-gate definitions,
6. Symbol table (optional), and
7. Comments (optional)

Header. The header consists of a single line
    `aag M I L O A`
where `aag` is an identifier for the ASCII variant of the AIGER format, `M` gives the maximum *variable index*, and `I, L, O, A` the number of inputs, latches, outputs, and AND gates, respectively.

In the rest of the specification, each input, latch, output, and AND gate is assigned a variable index i. To support negation, variable indices i are even numbers, and the negation of a variable can be referred to as $i+1$.

---

[4] `http://fmv.jku.at/aiger/`. Accessed February 2016.

[5] `http://fmv.jku.at/hwmcc/`. Accessed February 2016.

Variable index 0 is reserved for the constant truth value `false`, and accordingly 1 refers to `true`. In the following, all numbers that represent inputs, latches, outputs or AND-gates need to be smaller or equal to `2M+1`.

Input definitions. Every input definition takes one line, and consists of a single number (the variable index of the input). Inputs are never directly negated, so they are always represented by even numbers.

Latch definitions. Every latch definition takes one line, and consists of an even number (the variable index that represents the latch), followed by a number that defines which variable is used to update the latch in every step. Latches are assumed to have initial value 0.

Output definitions. Every output definition takes one line, and consists of a single number (representing a possibly negated input, latch, or AND-gate). For our class of (safety) problems, there is exactly one output, and safety conditions are encoded such that the circuit is safe if the output is always 0.

AND-gate definitions. Every AND-gate definition takes one line, and consists of three numbers. The first is an even number, representing the output of the AND-gate, and is followed by two numbers representing its (possibly negated) inputs.

Symbol table. The symbol table assigns names to inputs, latches, and outputs. It is optional, and need not be complete. Every line defines the name of one input, latch, or output, and starts with `i`,`l`,`o`, respectively, followed by the number of the input, latch, or output in the sequence of definitions (*not* the variable index of the input - so the first input is always defined by a line starting with `i0`, the first latch with `l0`). This is followed by an arbitrary string that names the variable.

3.2 Modified AIGER format for synthesis specifications

The SYNTCOMP format is a simple extension of the AIGER format for controller synthesis: we reserve the special string "`controllable_`" in the symbol table, and prepend it to the names of controllable input variables. All other input variables are implicitly uncontrollable.

The *synthesis problem defined by an extended AIGER file* is to find a circuit that supplies valuations for the controllable inputs, based on valuations of uncontrollable inputs and latches of the given circuit, such that the output always remains 0.

3.3 Output of synthesis tools in AIGER format

Starting from an input as defined in Section 3.2, we define when an AIGER file is a *solution* of the synthesis problem. Informally, the solution must contain the specification circuit, and must be verifiable by existing

model checkers that support the AIGER format. We give a more detailed definition in the following.

3.3.1 Syntactic correctness

Below we define how the input file can be changed in order to obtain a syntactically correct solution. Unless specified otherwise below, the output file must contain all lines of the input file, unmodified and in the same order.

Header. The original header line
    `aag M I L O A`
must be modified to
    `aag M' I' L' O A'`
where

- `I'` $= $ `I` $- c$
  (for c controllable inputs in the specification)
- `L'` $= $ `L` $+ l$
  (for l additional latches defined in the controller)
- `A'` $= $ `A` $+ a$
  (for a additional AND-gates defined in the controller)
- `M'` $= $ `I'` $+ $ `L'` $+ $ `A'`

The correct value for c can be computed from the symbol table of the input file, while correct values for l and a depend on the number of latches and AND-gates in the solution.

Inputs. Definitions for uncontrollable inputs remain unchanged. Definitions for controllable inputs are removed, and the corresponding variable indices have to be redefined either as new latches or AND-gates (see below).

Latches. No definitions of latches may be removed, but additional latches may be defined in the lines following the original latches.

Outputs. No definitions of outputs may be removed, no additional outputs may be defined.

AND-gates. No definitions of AND-gates may be removed, but additional AND-gates may be defined in the lines following the original AND-gates.

Global restrictions. All variable indices of controllable inputs have to be redefined exactly once, either as a new latch or as a new AND-gate. New latches and AND-gates may be defined using the remaining (uncontrollable) inputs, any latches, or newly defined AND-gates, but *not* original AND-gates.[6]

Symbol table and Comments. The symbol table remains unchanged. Comments may be removed or modified at will.

---

[6] The reason for disallowing original AND-gates is that we want the controller to work only based on the *state* of the given circuit (i.e., values of latches), and the uncontrollable inputs. Original AND-gates can be duplicated in the controller if necessary.

3.3.2 Semantic Correctness

All input files will have the same structure as *single safety property* specifications used in HWMCC. In particular, this means that there is only one output, and the system is safe if and only if this output remains 0 for any possible input sequence.

Any output file satisfying the syntactical restrictions described in Section 3.3.1 is an AIGER file. It is *correct* if for any input sequence (of the uncontrollable inputs), the output always remains 0. We say that it is a *solution* to the synthesis problem defined by the input file if it is successfully model checked by an AIGER-capable model checker within a determined time bound.

4 Benchmarks

The benchmark set for the first SYNTCOMP consisted of 569 benchmark problems overall, out of which 390 are realizable and 179 unrealizable.[7] Most of the benchmarks existed before in other formats, and have been translated to our new format. The full set of benchmarks used in SYNTCOMP 2014 is available in directory `Benchmarks2014` of our public Git repository at `https://bitbucket.org/swenjacobs/syntcomp/`. In the following, we first explain how benchmarks have been collected, translated and tested, and then describe the different sets of benchmarks.

4.1 Collection of Benchmarks

One of the major challenges for the first SYNTCOMP was the collection of benchmarks in the extended AIGER format. Following the decision to use this format, a call for benchmarks was sent to the synthesis community. Many synthesis tools have their own benchmark set, but none of them previously used the SYNTCOMP format, and therefore such benchmarks had to be translated. Since we restrict to safety specifications currently, such a translation usually involves a safe approximation of liveness by safety properties, and results in a family of benchmark instances for different precision of the approximation.

Generation and Translation of Benchmarks. One method for obtaining benchmarks in AIGER format is based on a translation from LTL specifications, together with a reduction to a bounded synthesis problem, as used in Acacia+[8] [14,33]. The idea is to 1) translate the negation of the LTL formula into a universal co-Büchi automaton, 2) strengthen this automaton into a universal

k-co-Büchi automaton that accepts a word w if and only if all the runs on w visit rejecting states at most k times — such an automaton defines a safety objective and can be easily made deterministic. 3) Finally, a safety game is obtained by encoding succinctly this deterministic safety automaton as an AIGER specification. We thus obtain a family of benchmark instances, one for each valuation of k. If the original LTL specification is realizable, then the resulting benchmark instance will be realizable for sufficiently large k. This translation from LTL to AIGER has been implemented by Guillermo A. Pérez in the `ltl2aig` routine[9].

Another successful way of obtaining benchmarks was to start from Verilog code, and use a toolchain composed of the `vl2mv` routine of the VIS system[10] [16], followed by translation to AIGER (and optimization) by ABC[11] [17], and from binary AIGER format to ASCII format by the `aigtoaig` routine from the AIGER tool set[12]. Liveness properties can be approximated by safety properties, and we obtain a family of benchmark instances for different approximations. Such an approximation is explained in more detail in Section 4.3. This approach will be called the *Verilog toolchain* below.

Finally, a number of benchmarks have been obtained by translation from structured specifications for the generalized reactivity(1) game solver SLUGS[13]. The term "structured" in this context refers to support for constraints over (non-negative) integer numbers, which are automatically compiled into a purely Boolean form. The purely Boolean generalized reactivity(1) safety specification is then translated into a monitor automaton in AIGER format, which is finally optimized using the ABC toolset by applying the command sequence `rewrite`. We will call this approach the *SLUGS toolchain* below.

Testing and Classification of Benchmarks. To test the resulting benchmarks, we fed them to our reference implementation Aisy[14], and compared the produced solution to the expected result. Since our reference implementation is not as efficient as the participants of the competition, a significant number of benchmarks was only solved during the competition, but not in our initial tests. Those that were not solved were classified into realizable and unrealizable according to informed guesses of the benchmark authors. During the competition, this resulted in 3 problem instances being re-classified from unrealizable to realizable, or vice versa.

---

[7] Numbers regarding realizability are to the best of our knowledge. The realizability status has not been verified for all benchmark instances.

[8] `http://lit2.ulb.ac.be/acaciaplus/`. Accessed February 2016

[9] `https://github.com/gaperez64/acacia_ltl2aig`. Accessed February 2016.

[10] `http://vlsi.colorado.edu/~vis/`. Accessed February 2016.

[11] `http://www.eecs.berkeley.edu/~alanmi/abc/`. Accessed February 2016.

[12] `http://fmv.jku.at/aiger/`. Accessed February 2016.

[13] `http://github.com/ltlmop/slugs`. Accessed February 2016.

[14] `https://bitbucket.org/art_haali/aisy-classroom`. Accessed February 2016.

4.2 Toy Examples

These benchmarks are based on original Verilog specifications that have been translated to AIGER using the Verilog toolchain. The set includes specifications of basic circuits like adders, bit shifters, multipliers, and counters. Additionally, it contains some specifications with typically very simple properties, e.g., that outputs must match inputs, or that the XOR of inputs and outputs must satisfy some property. All examples are parameterized in the bit-width of the controllable and uncontrollable inputs, ranging between 2 and 128 bits on some examples, and for each example there are two versions, using the optimizing and non-optimizing translation by ABC, respectively. Overall, this set contains 138 benchmarks.

All AIGER files contain the original Verilog code, as well as the commands used to produce the AIGER file, in the comments section. This set of benchmarks was provided by Robert Könighofer.

4.3 Generalized Buffer

The well-known specification of an industrial generalized buffer was developed by IBM and subsequently used as a synthesis benchmark for Anzu [45] and other tools. It is parameterized by the number of senders which send data to two receivers. The buffer has a handshake protocol with each sender and each receiver. A complete Genbuf consists of a controller, a FIFO, and a multiplexer. In this benchmark, the FIFO and multiplexer are considered as part of the environment, and the controller is synthesized. As a synthesis case study for Anzu, it has been explained in detail by Bloem et al. [11]. Robert Könighofer translated these benchmarks to AIGER, as explained in the following.

Liveness-to-Safety Translation. For Anzu, the Genbuf benchmark contains Büchi assumptions $\{A_1, \ldots, A_m\}$ that are satisfied if all state sets A_i are visited infinitely often, and Büchi guarantees $\{G_1, \ldots, G_n\}$ requiring that all G_j are visited infinitely often if all assumptions are satisfied. Three different translations into safety specifications were performed. Translation "c" (for *counting*) applies the well-known counting construction: A modular counter $i \in \{0, \ldots, m\}$ stores the index of the next assumption. If an accepting state $s \in A_i$ of this next assumption is visited, the counter is incremented modulo $m + 1$. If i has the special value 0, it is always incremented. The same counting construction is applied to the Büchi guarantees with counter $j \in \{0, \ldots, n\}$. Finally, a third counter r is used to enforce a minimum ratio between the progress in satisfying guarantees and assumptions: Whenever j is incremented, r is reset to 0. Otherwise, if $i = 0$, then r is incremented. If r ever exceeds some bound k, then BAD is set. A controller enforcing that BAD cannot become 1 thus also enforces

that all G_j are visited infinitely often if all A_i are visited infinitely often. Translation "b" (for *bitwise*) is similar but uses one bit per assumption and guarantee instead of a counter. It thus avoids imposing an (artificial) order between properties. Translation "f" (for *full set*) is similar to "b" but resets r only if all guarantees have been seen in a row (rather than only the next one).

Translation to AIGER. Anzu comes with scripts to construct Genbuf benchmark instances with different numbers of senders. These scripts were modified to output a Verilog representation, and from there the Verilog toolchain was used to obtain benchmarks in AIGER format. The final specification is parameterized in 1) the number of senders which send data, 2) the type (c, b or f) and the bound k of the liveness-to-safety translation, and 3) whether or not ABC optimizations are used in the translation. All AIGER files contain the original Verilog code (which in turn contains the Anzu specification it was created from), as well as the commands used to produce the AIGER file, in the comments section. Overall, this set contains 192 benchmark instances.

4.4 AMBA Bus Controller

This is a specification of an arbiter for the AMBA AHB bus, based on an industrial specification by ARM. Like the Genbuf case study, it has been used as a synthesis benchmark for Anzu [45] and other tools. It is parameterized with the number of masters that can access the bus and have to be coordinated by the arbiter. The AMBA AHB bus allows masters to request different kinds of bus accesses, either as a single transfer or as a burst, where a burst can consist of either a specified or an unspecified number of transfers. Besides correct modeling of these different forms of accesses, the specification requires responses to all requests (that are not eventually lowered), as well as mutual exclusion of bus accesses by different masters. As a synthesis case study for Anzu, it has been explained in detail by Bloem et al. [10].

The Anzu specification has been translated by Robert Könighofer to AIGER format using the Verilog toolchain in the same way as for the Genbuf benchmark. Instances are parameterized in the number of masters, and (as for Genbuf) the type (c, b or f) and the bound k of the liveness-to-safety translation, as well as whether or not ABC optimizations are used in the translation. All AIGER files contain the original Verilog code (which in turn contains the Anzu specification it was created from), as well as the commands used to produce the AIGER file, in the comments section. Overall, this set contains 108 benchmarks.

4.5 LTL2AIG Benchmarks

This set contains several benchmarks provided in the Acacia+ tool package [14], translated into AIGER format using the `ltl2aig` routine. The set includes:

- 50 benchmarks from the test suite included with the synthesis tool LILY[15] [44], with specifications of traffic lights and arbiters in different complexity (25 original examples, each with 2 different choices of k).
- 4 versions of the Genbuf case study, but in a much more challenging form than the specification mentioned in Section 4.3[16] This version is only specified for 2 senders and 2 receivers, and for 4 different choices of k.
- 5 versions of a load balancer case study, originally presented with synthesis tool UNBEAST[17] [28].
- 23 benchmarks that use the synthesis tool to obtain a deterministic Büchi automaton for the given LTL specification (if possible), and
- 18 benchmarks for a similar conversion from LTL to deterministic parity automata. The latter two conversions are mentioned as applications of synthesis procedures by Kupferman and Vardi [46].

Overall, this set contains 100 benchmarks.

4.6 Factory Assembly Line

This benchmark models an assembly line with multiple tasks that need to be performed on the objects on the conveyor belt. The specification models a number of robot arms (fixed to 2), a number n of objects on the conveyor belt, and a number m of tasks that may have to be performed on each object before it leaves the area that is reachable by the arms. The belt moves after a fixed number k of time steps, pushing all objects forward by one place, and the first object moves out of reach of the arms (while a new object enters at the other end of the belt). The arms are modeled such that they cannot occupy the space above the same object on the belt, and can move by at most one position per time step. In particular, this means that they cannot pass each other. Whenever an arm is in the same position as an object that has unfinished tasks, it can perform one task on the object in one time step. Usually, the assumption is that at most $m-1$ of the m tasks need to be performed on any object, but there may be a fixed number c of *glitches* in an execution of the system, which means that an object with m open tasks is pushed onto the belt.

---

[15] `http://www.iaik.tugraz.at/content/research/opensource/lily/`. Accessed February 2016.

[16] We conjecture that this version is more challenging because it is based on a large LTL specification, which is translated to a single, very big Büchi automaton in the first step of the `ltl2aig` routine. This results in a circuit that is much more complex than the ones from Section 4.3.

[17] `http://www.react.uni-saarland.de/tools/unbeast/`. Accessed February 2016.

This specification has been translated by Rüdiger Ehlers from original benchmarks for the SLUGS GR(1) synthesis tool, using the SLUGS toolchain. Overall, this set contains 15 benchmarks, for different values of m (3 to 7), n (3 to 6), k (1 to 2) and c (0 to 11).

4.7 Moving Obstacle Evasion

This benchmark models a controller for a robot that moves on a quadratic grid of parametric size m, and has to avoid colliding with a moving obstacle (of size 2×2 grids). In any time step, the robot and the obstacle can only move by at most one grid in x and y direction. Additionally, the obstacle can usually only move at most every second time step. However, as in the assembly line benchmarks, there may be a fixed number c of glitches in an execution of the system, which in this case means that the obstacle moves even though it has already moved in the immediately preceding time step.

This specification has been translated by Rüdiger Ehlers from original benchmarks for the SLUGS GR(1) synthesis tool, using the SLUGS toolchain. Overall, this set contains 16 benchmarks, for different values of m (8 to 128) and c (0 to 60).

5 Rules

The rules for SYNTCOMP were inspired by similar competitions such as the SAT competition and the HWMCC. The basic idea is that submitted tools are evaluated on a previously unknown set of benchmarks, without user intervention. A simple ranking of tools can be obtained by checking only the correctness of solutions, and counting the number of problem instances that can be solved within a given timeout. However, the goal of synthesis is to obtain implementations that are not only correct, but also efficient. Therefore, we also considered refined rankings based on the *quality* of the produced solutions, measured by the size of the implementation.

Tracks. The competition was separated into two tracks: the *realizability track* which only required a binary answer to the question whether or not there exists a circuit that satisfies the given specification, and the *synthesis track* which was only run on realizable benchmarks, and asked for a circuit that implements the given specification. The motivation for this split was again to have a low entry-barrier for tool creators, as an efficient realizability checker can be implemented with less effort than a full synthesis tool that produces solutions and optimizes them for size. Indeed, 2 out of 5 submitted tools make use of this split and only supply a realizability checker, and these two tools solve more problems in the realizability track than any of the full synthesis tools.

Subtracks. Each track was divided into a *sequential* subtrack, where tools can use only one core of the CPU,

and a *parallel* subtrack, allowing tools to use multiple cores in parallel. The decision to have both sequential and parallel execution modes was based on the expectation that parallelization would often be trivial—i.e., a number of different but largely independent strategies running in parallel.[18] Therefore, we also wanted to evaluate tools in sequential execution mode in order to measure and identify the single best strategy.

5.1 Entrants

We asked for synthesis tools to be supplied in source code, licensed for research purposes, and we offered to discuss possible solutions if this restriction was a problem to any prospective participant. This was not the case for any of the research groups that contacted us regarding the competition. The organizers reserved the right to submit their own tools and did so in the form of BASIL, implemented by R. Ehlers, and Demiurge, implemented in part in the research group of R. Bloem. We encouraged participants to visit SYNT and CAV for the presentation of the SYNTCOMP results, but this was not a requirement for participation.

We allowed up to 3 submissions per author and subtrack, where submissions are considered to be different if source code, compilation options or command line arguments are different. This limit was chosen to allow some flexibility for the tool creators, while avoiding the flooding of the competition with too many different configurations of the same tool. All tools must support the input and output format of SYNTCOMP, as defined in Section 3. Additionally, each entrant to SYNTCOMP was required to include a short system description.

The organizers commited to making reasonable efforts to install each tool but reserved the right to reject entrants where installation problems persisted. This was not the case for any of the entrants. Furthermore, in case of crashes or wrong results we allowed submission of bugfixes if possible within time limits. In one case, a bugfix was submitted that resolved a number of solver crashes that only appeared during the competition runs.

5.2 Ranking

In both the realizability and the synthesis track, competition entrants were ranked with respect to the number of solved problems. Additionally, we consider a more fine-grained *relative ranking* that distributes points for each benchmark according to the relative performance of tools, measured either in the time needed to find a solution, or the size of the solution. A drawback of this relative ranking is that it does not allow easy comparison to tools that did not participate. As an alternative that resolves this problem, we additionally give a *quality ranking* for the synthesis track that compares the size of the provided solution to a reference size.[19]

For all rankings, a timeout (or no answer) gives 0 points. A punishment for wrong answers was not necessary, since the full set of benchmarks was made available to the participants one month before the submission of solvers.

Correctness and Ranking in Realizability Track. For the realizability track, the organizers and benchmark authors took responsibility for determining in advance whether specifications are realizable or unrealizable, by using knowledge about how the benchmarks were generated. When in doubt, a majority vote between all tools that solved a given benchmark was used to determine the correct outcome.[20]

In addition to a ranking based on the number of solved problem instances, tools were evaluated with a relative ranking based on the time needed to come to the solution, where the tool with the smallest time earns the highest rank (see below). For the sequential subtrack, tools were ranked with respect to CPU time, while for the parallel subtrack we ranked tools with respect to wall-clock time.

Correctness and Ranking in Synthesis Track. In the synthesis track, correctness of solutions was assessed by checking both syntactical and semantical correctness. Syntactical correctness means conformance to our output format defined in Section 3, which was checked by a separate syntax checker. Semantical correctness was tested by a model checker (`iimc`[21], based on the IC3 algorithm [15]), which had to terminate within a separate time bound for the result to be considered correct. As in the realizability track, there is a ranking with respect to the number of solved problem instances, as well as a relative ranking. The latter is in this case based on the size of solutions, given by the number of AND-gates in the resulting circuit. In addition, we provide a quality ranking that awards points for every solution, based on a comparison of the solution size to a reference size (see below).

Relative Ranking. For every benchmark, all tools that provide a correct solution are ranked with respect to the metric (time or size), and each tool obtains points based on its rank. In detail: if k benchmarks are used in the

---

[18] In particular, non-trivial parallelization is difficult for BDD-based tools, since none of the existing parallel BDD packages supports all features needed for the optimizations mentioned in Section 2.3.

[19] The quality ranking was devised for the second SYNTCOMP and was applied to the results of the first competition only after the presentation of results at SYNT and CAV 2014.

[20] This rule only had to be used in one instance, where a benchmark was solved by only one tool, and was reported to be realizable although unrealizable was the expected outcome. In our analysis it turned out that the tool was correct, and the initial classification as unrealizable was wrong.

[21] http://ecee.colorado.edu/wpmu/iimc/. Accessed February 2016.

track, then $1000/k = p$ points are awarded per benchmark. If n tools solve the benchmark, then the points for that benchmark are divided into $\Sigma_{i=1}^n i = f$ fractions, and the tool which is at rank m will get $\frac{n-m+1}{f} \cdot p$ points for this benchmark.

Quality Ranking. In the *quality ranking*, solutions are awarded points depending on the size $size_{new}$ of the solution and a reference size $size_{ref}$. The number of points for a solution is

$$2 - \log_{10}\left(\frac{size_{new}}{size_{ref}}\right).$$

That is, a solution that is of size $size_{ref}$ gets 2 points; a solution that is bigger by a factor of 10 gets 1 point; a solution that is bigger by a factor of 100 (or more) gets 0 points; and similarly for solutions that are smaller than $size_{ref}$.

Since for the first competition we do not have reference solutions for any of the problem instances, we use the smallest size of any of the solutions of this competition as the reference size. In future competitions, or for comparison of tools that did not participate, the size of the smallest solution that has been provided in any of the competitions before can be used.

6 Participants

Five systems were entered into the first SYNTCOMP. In the following, we give a brief description of the methods implemented in each of these systems. For the BDD-based tools, Table 1 shows which of the optimizations from Section 2.3 are implemented in which tool.

6.1 AbsSynthe: an abstract synthesis tool

AbsSynthe was submitted by R. Brenguier, G. A. Pérez, J.-F. Raskin, and O. Sankur from Université Libre de Bruxelles. AbsSynthe implements a BDD-based synthesis approach and competed in all subtracks.

Synthesis algorithms. AbsSynthe implements different BDD-based synthesis algorithms, with and without abstraction, described in more detail in [18]. All algorithms use the BDD package CUDD [60], with automatic BDD reordering using the sifting heuristic.

The *concrete algorithm with partitioned transition relation* (C-TL) implements BDD-based synthesis with partitioned transition relation and direct substitution of state variables with BDDs. In addition, when computing $\mathsf{UPRE}(S(L'))$, then the transition functions f_l of all latches are first restricted to $\neg S(L)$, effectively only computing the uncontrollable predecessors which are not already in $S(L)$. These new states are then joined to $S(L)$, which gives the same result as the standard UPRE computation.

The *basic abstract algorithm* (A) implements synthesis with a precomputed (monolithic) abstract transition relation, and some additional optimizations.

The *alternative abstract algorithm* (A-TL) avoids using a precomputed transition relation by implementing abstract operators for post-state computation.

AbsSynthe was intended to compete in these different configurations. However, due to a miscommunication between tool authors and competition organizers, the necessary command line parameters were not used, such that only one configuration participated, namely (C-TL). Unfortunately, this error was discovered too late to run the additional configurations before the presentation of results at CAV 2014.

However, as mentioned in [18], the abstraction-based methods overall performed worse than the concrete algorithm (C-TL), and thus the fastest configuration did participate in the competition.

Strategy extraction. Strategy extraction in AbsSynthe uses the co-factor-based approach described in Section 2.3, including the additional forward reachability check. When extracting the circuit, every AIG node constructed from the BDD representation is cached in order to avoid duplicating parts of the circuit.

Implementation, availability. AbsSynthe is written mostly in Python, and depends only on a simple AIG library (fetched from the AIGER toolbox[22]) and the BDD package CUDD[23]. The source code is available at `https://github.com/gaperez64/AbsSynthe`.

6.2 BASIL: BDD-based safety synthesis tool

BASIL was submitted by R. Ehlers from the University of Bremen, and implements a BDD-based synthesis approach. BASIL competed in all subtracks.

Synthesis algorithm. BASIL implements a BDD-based synthesis algorithm, based on the BDD package CUDD. It uses automatic reordering of BDDs with the *sifting* heuristic, reconfigured in order to optimize more greedily. In contrast to all other BDD-based tools in the competition, it does not use a partitioned transition relation. It does however use a technique similar to direct substitution, regarding latches that are always updated by the value of an input variable: BDD variables that represent such inputs are double-booked as both an input and a post-state variable of the latch, and therefore need not be explicitly encoded into the transition relation. Additionally, when building the transition relation it eagerly deletes BDDs that are only used as intermediate values as soon as they are not needed anymore. This is the case

---

[22] `http://fmv.jku.at/aiger/`. Accessed February 2016.

[23] `http://vlsi.colorado.edu/~fabio/CUDD/`. Accessed February 2016.

Table 1. Optimizations implemented in BDD-based Tools.

Technique	AbsSynthe	Basil	Realizer	Simple BDD Solver
automatic reordering	x	x	x	x
eager deallocation of BDDs		x		x
direct substitution	x	(x)	x	x
partitioned transition relation	x		x	x
simultaneous conjunction and abstraction				x
co-factor based extraction of winning strategies	x	x	N/A	N/A
forward reachability analysis	x	x	N/A	N/A
ABC minimization		x	N/A	N/A
additional optimizations (see tool descriptions)	x	x	x	x

if a gate A is not used as a controllable input or an input to a latch, and all nodes that depend on A have been processed.

Strategy extraction. Basil computes strategies with the co-factor-based approach from Section 2.3, including forward reachability analysis and ABC minimization.

As an additional optimization, during strategy extraction the output bit BDDs are reduced in size by applying *LICompaction* [38]: A joint BDD for all output bit BDDs is built and then, in a round-robin fashion over the outputs, the size of the joint BDD is reduced by changing a part of it that describes the behavior of a single output bit in a way that makes the overall BDD smaller, but yields behavior that is contained in the most general strategy for winning the game. In order to minimize the care set for this operation, a reachable-state computation is performed before every step. When no further size reduction is found to be possible, or some timeout has been reached, optimization by *LICompaction* is aborted.

Implementation, availability. Basil is implemented in C++ and depends on the BDD package CUDD, as well as (optionally) ABC[24] for strategy minimization. It is currently not publicly available.

6.3 Demiurge

Demiurge was submitted by R. Könighofer from Graz University of Technology and M. Seidl from Johannes-Kepler-University Linz. Demiurge implements incremental SAT- and QBF-based synthesis as described in Section 2.4, as well as template-based synthesis with QBF solving as described in Section 2.5. Demiurge competed in all subtracks.

Synthesis algorithms. Demiurge implements different synthesis algorithms in different back-ends, described in more detail in [13].

The *learning-based back-end* uses the incremental synthesis approach to compute a winning region based on two competing SAT solvers to compute and generalize states to be removed from the winning region (algorithm

LearnSat from [13] with optimization RG enabled, but optimization RC disabled). Minisat version 2.2.0 is used as underlying SAT solver.

The *parallel back-end* implements the same method with three threads refining the winning region in parallel. Two threads perform the work of the learning-based back-end, one using Minisat 2.2.0 and the other using Lingeling ats. The third thread generalizes existing clauses of the winning region further by trying to drop more literals. Using different solvers in the threads is beneficial because the solvers can complement each other, sometimes yielding a super-linear speedup [13].

The *template-based back-end* uses a QBF solver to compute a winning region as instantiation of a template for a CNF formula over the state variables. For SYNTCOMP, DepQBF 3.02 is used as QBF solver via its API. Bloqqer, extended to preserve satisfying assignments [58], is used as QBF preprocessor.

Demiurge contains more back-ends that are either experimental or did not turn out to be particularly competitive, and therefore did not enter the competition. This includes a re-implementation of the technique of Morgenstern et al. [51], and an approach based on a reduction to Effectively Propositional Logic (EPR). Details can be found in [13].

Strategy extraction. Demiurge provides several methods for computing strategies from the winning region. The algorithm used in the competition uses a computational learning approach as proposed in [30], but implemented with incremental SAT solving or incremental QBF solving instead of BDDs. In terms of [9], it uses the SAT-based learning method without the dependency optimization, with Lingeling ats as SAT solver. ABC minimization is used in a post-processing step.

Implementation, availability. Demiurge is implemented in C++, and depends on a number of underlying reasoning engines, some of them mentioned above. Because of its modular architecture, Demiurge is easily extendable with new algorithms and optimizations (cf. [9]), thus providing a framework for implementing new synthesis algorithms and reducing the entry barrier for new research on SAT- and QBF-based synthesis algorithms and optimizations.

---

[24] http://www.eecs.berkeley.edu/~alanmi/abc/.   Accessed February 2016.

Demiurge is available under the GNU LGPL license (version 3) at `http://www.iaik.tugraz.at/content/research/design_verification/demiurge/`.

6.4 REALIZER– CUDD Based Safety Game Solver

Realizer was submitted by L. Tentrup from Saarland University, Saarbrücken. Realizer implements BDD-based realizability checking, and competed in both realizability subtracks. It does not support extraction of strategies.

Synthesis algorithms. Realizer is based on BDD package CUDD, and uses automatic reordering of BDDs with the *lazy sift* reordering scheme. When building the BDDs that represent the transition relation, it uses a temporary hash table to save the BDDs for AND gates in the AIG. Before starting the fix-point algorithm, it builds the basic data structures used in the fixed point calculation, like the arrays mapping the current state variable to the next (primed) state variable or the BDD cubes used for the existential and universal abstraction.

The actual fix-point algorithm is implemented in two variants, differing only in the way they handle the forced predecessor function: one variant uses a monolithic transition relation, while the other uses a partitioned transition relation. Both variants use direct substitution.

The *variant with partitioned transition relation* overall performed better in preliminary experiments, so only this one was entered into the competition in the sequential realizability track. Since on some examples the other variant performed better, the *parallel version* uses both variants running (independently) in parallel.[25]

Implementation, availability. Realizer is written in Python and uses the BDD library CUDD in version 2.4.2 with the corresponding Python bindings PyCUDD in version 2.0.2. It is currently not publicly available.

6.5 Simple BDD Solver

Simple BDD Solver was submitted by L. Ryzhyk from NICTA, Sydney and the Carnegie Mellon University, Pittsburgh, and A. Walker from NICTA, Sydney. Simple BDD Solver implements BDD-based realizability checking, and only competed in the sequential realizability subtrack. It does not support extraction of strategies.

Synthesis algorithm(s). Simple BDD Solver is a substantial simplification of the solver that was developed for the Termite project[26], adapted to safety games given in the AIGER format. It uses the BDD package CUDD, with dynamic variable reordering using the sifting algorithm, and eager deallocation of BDDs. Furthermore, it uses a partitioned transition relation, direct substitution, and simultaneous conjunction and abstraction.

Additionally, it uses an alternative form for the fixpoint computation of UPRE that avoids creating the BAD latch and simplifies the quantification structure:

The fixpoint formula (1) in Section 2.3 is equivalent to

$$\mu S(L).\ \exists X_u \forall X_c \exists L'.$$
$$\big(S(L') \wedge T(L, X_u, X_c, L')\big) \vee \mathsf{BAD}'.$$

To avoid introducing the latch for BAD, we substitute BAD' with the update function for BAD - an expression $\neg SAFE(L, X_u, X_c)$ over latches and inputs. This results in:

$$\mu S(L).\ \exists X_u \forall X_c \exists L'.$$
$$\big(S(L') \wedge T(L, X_u, X_c, L')\big) \vee \neg SAFE(L, X_u, X_c).$$

Then, quantifiers are re-arranged to

$$\mu S(L).\ \exists X_u \forall X_c.$$
$$\big(\exists L'.\ S(L') \wedge T(L, X_u, X_c, L')\big) \vee \neg SAFE(L, X_u, X_c),$$

with the safety condition outside of the innermost existential quantification. With this formula for the fixpoint, simultaneous conjunction and abstraction can be used on the left-hand side of the disjunction, and we avoid to build the potentially large BDD of the conjunction in the left-hand side at every iteration.

Furthermore, the tool implements a variant of the fixpoint algorithm with an abstraction-refinement loop inspired by [26]. Since this variant has not been found to be competitive on the set of competition benchmarks, it has not been entered into the competition.

Implementation, availability. Simple BDD solver is written in the Haskell functional programming language. It uses the CUDD package for BDD manipulation and the Attoparsec[27] Haskell package for fast parsing. Altogether, the solver, AIGER parser, compiler and command line argument parser are just over 300 lines of code. The code is available online at: `https://github.com/adamwalker/syntcomp`.

7 Execution

SYNTCOMP 2014 used a compute cluster of identical machines with octo-core Intel Xeon processors (2.0 GHz) and 64 GB RAM, generously provided by Graz

---

[25] Analysis of results and subsequent inspection of the source code by the tool author showed that due to a bug the parallel version did not work as intended, and instead used two threads with identical strategy. As can be seen in the results section, this lead to a decreased performance overall.

[26] `http://termite2.org`. Accessed February 2016.

[27] `https://hackage.haskell.org/package/attoparsec`. Accessed February 2016.

University of Technology. The machines are running a GNU/Linux system, and submitted solvers were compiled using GCC version 4.7. Each node has a local 400 GB hard drive that can be used as temporary storage.

The competition was organized on the EDACC platform [4] developed for the SAT Competitions [43]. EDACC directly supports the definition of subtracks with different benchmark sets, different solver configurations, verification of outputs, and automatic distribution of jobs to compute nodes. During the competition, a complete node was reserved for each job, i.e., one synthesis tool (configuration) running one benchmark. This ensures a very high comparability and reproducibility of our results. Olivier Roussel's `runsolver` [56] 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 per benchmark (CPU time for sequential subtracks, wall time for parallel subtracks) was set. The timeout for each task in any subtrack was 5000 seconds (CPU time or wall time, respectively). The queueing system in use is TORQUE[28].

Some solvers did not conform completely with the output format specified by the competition, e.g. because extra information was displayed in addition to the specified output. For these solvers, small wrapper scripts were used to execute them, filtering the outputs as to conform to the specified format.

Validity of results. Beyer et al. [7] recently noted that `runsolver`, along with a number of other benchmarking tools, has certain deficits that endanger the validity of results. In particular, the CPU time of child processes may not be measured correctly. First, note that CPU time is only relevant for our results in the sequential subtracks, where tools are restricted to a single CPU core. Furthermore, for the participants of SYNTCOMP we note that the only child processes (if any) are the reasoning engines for BDDs and SAT or QBF formulas. Since these reasoning engines take up almost all of the CPU time in solving synthesis tasks, a comparison of CPU time to the recorded wall time would most probably reveal measurements that exclude child processes. This was not the case for our results.

8 Experimental Results and Analysis

We present the results of SYNTCOMP 2014, separated into realizability and synthesis tracks, followed by some observations on the state of the art. 5 tools entered the competition and ran in 8 different configurations in the 4 tracks of the competition.[29] All of

the results can be viewed online in our EDACC system at `https://syntcompdb.iaik.tugraz.at/2014/experiments/`. Furthermore, the full experimental data, including problem instances, executable code of the solvers, logfiles of executions, solutions produced by solvers, and executable code for verifying the solutions is available in directory `ExperimentalData2014` of our public Git repository at `https://bitbucket.org/swenjacobs/syntcomp/`.

8.1 Realizability Track

In the realizability track, tools were run on the full set of 569 benchmarks. All 5 tools that entered SYNTCOMP competed with at least one configuration in the sequential subtrack, and 4 of them also competed in the parallel subtrack.

Sequential Subtrack. The sequential realizability track had 6 participants: AbsSynthe, BASIL, Realizer and Simple BDD Solver competed with one configuration each, whereas Demiurge competed with two different configurations. Table 2 contains the number of instances solved within the timeout per tool, the number of instances solved uniquely by a solver, and the accumulated points per tool according to our relative ranking scheme.

No tool could solve all 569 benchmarks, and 13 benchmarks were not solved by any of the tools within the timeout. 12 benchmarks were solved uniquely by one tool:

- BASIL: 4 versions of the factory assembly benchmarks (size 5x5 and 7x5, each with 10 and 11 errors)
- Realizer: `gb_s2_r2_1_UNREAL`[30]
- Demiurge (templ): `mult1x` with $x \in \{2, 3, 4, 5, 6\}$, `stay18n` and `stay20n`

Table 3 gives an overview of benchmark instances that were solved uniquely or not solved at all, for all subtracks of the competition.

Furthermore, Figure 2 gives an overview of solved instances by participant and benchmark classes (see Section 4), and Figure 3 a cactus plot for the time needed by each tool to solve the benchmarks.

Analysis. Table 2 and Figure 2 show that the BDD-based tools AbsSynthe, BASIL, Realizer and Simple BDD Solver are very close to each other when only comparing the number of instances solved. Furthermore, for the Amba and Genbuf benchmarks, some tools solve all instances in the benchmark set, i.e., we would need more difficult instances to distinguish which tool is better for these classes.

---

was always started in the same configuration. The results presented here for the relative ranking differ from those presented at CAV 2014 in that only one of the three identical configurations of AbsSynthe is counted in the sequential tracks.

[30] This benchmark was found to be realizable by the tool, although it was classified as unrealizable by the benchmark authors. Our analysis confirmed it to be realizable.
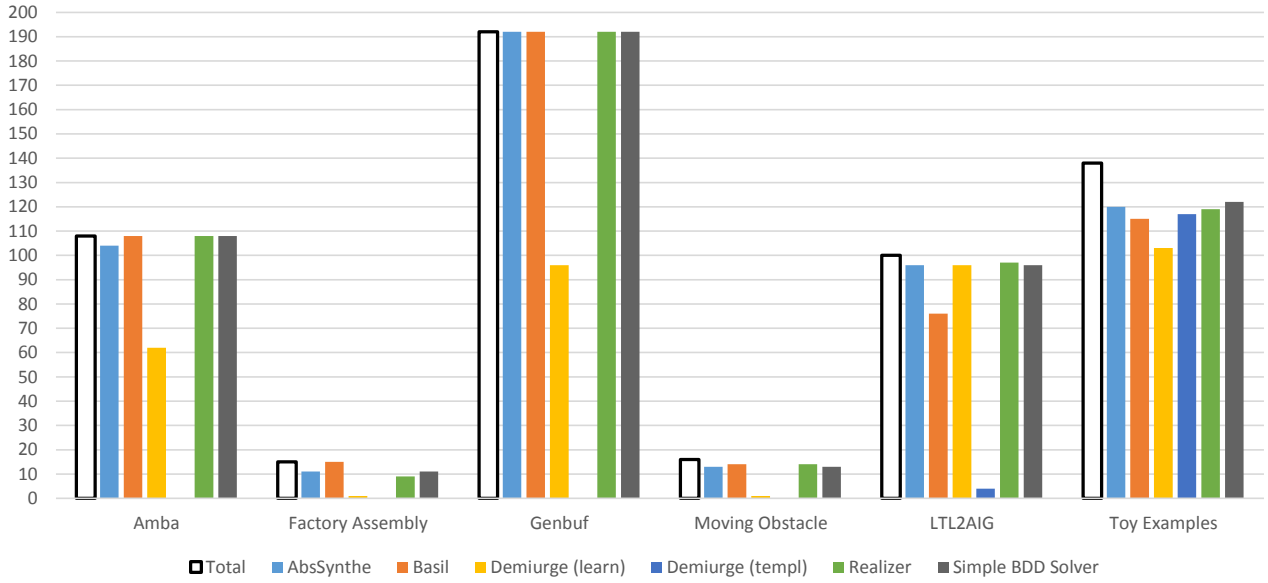
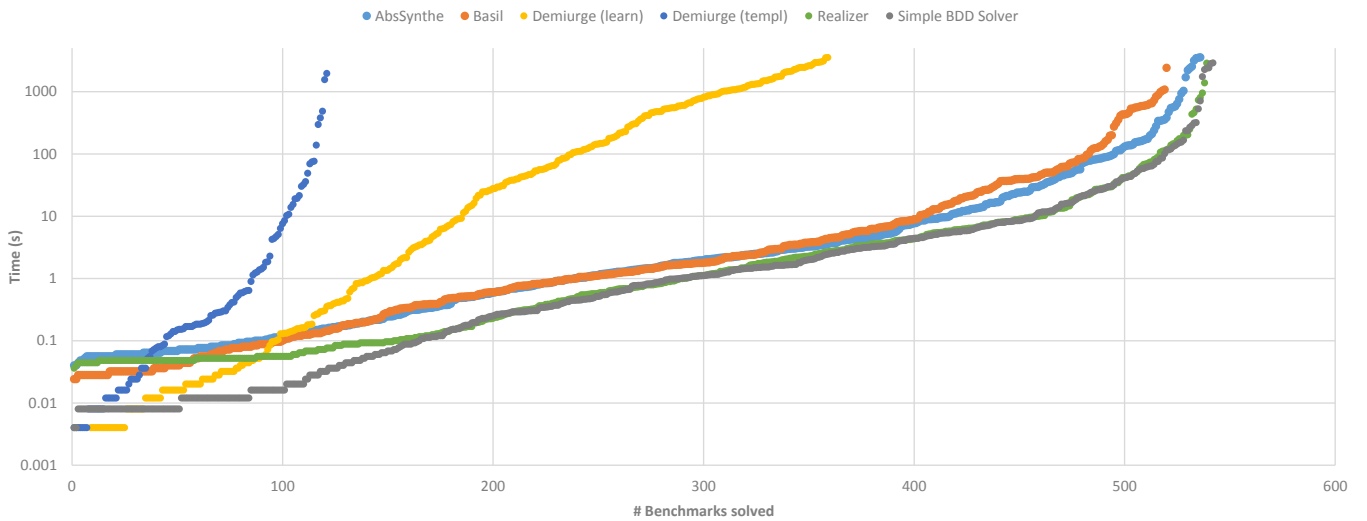**Fig. 2.** Sequential Realizability Track, Solved Instances by Category

Fig. 3. Sequential Realizability Track, Cactus Plot

Table 2. Results of the Sequential Realizability Track

Tool	Solved	Unique	Relative
Simple BDD Solver	**542**	0	**262**
Realizer	539	1	229
AbsSynthe	536	0	144
Basil	520	4	209
Demiurge (learn)	359	0	209
Demiurge (templ)	121	**7**	90

The best result in each column is in bold.

Regarding the SAT- and QBF-based synthesis approaches, Demiurge (learn) solves about as many of the

LTL2AIG benchmarks as the best BDD-based tools, and almost as many of the Toy Examples. For AMBA and Genbuf Benchmarks, Demiurge (learn) solves only about half as many benchmarks, and for the Moving Obstacle and Factory Assembly benchmarks can only solve one in each case. Finally, Demiurge (templ) can solve a very good number of the Toy Examples, and even solves 7 of them uniquely. However, it solves only very few of the LTL2AIG benchmarks, and none of the others.

As can be seen in Figure 3, most tools have a steep degradation with higher complexity of benchmarks, i.e., between 90% and 95% of the benchmarks that can be

Table 3. Benchmark instances that were solved uniquely or not solved at all in at least one subtrack. "Yes" means solved by more than one tool, "Yes*" means uniquely solved. "-" means that this benchmark instance was not tested.

Benchmark	Solved in Seq. Realizability	Solved in Par. Realizability	Solved in Seq. Synthesis	Solved in Par. Synthesis
amba8c7y	Yes	Yes	No	No
amba9c5y	Yes	Yes	Yes	Yes*
amba10c5y	Yes	Yes	Yes	Yes*
cnt30n	No	No	-	-
cnt30y	No	No	No	No
factory_assembly_5x5_2_10errors	Yes*	Yes*	No	No
factory_assembly_5x5_2_11errors	Yes*	Yes*	No	No
factory_assembly_7x5_2_10errors	Yes*	Yes*	No	No
factory_assembly_7x5_2_11errors	Yes*	Yes*	No	No
gb_s2_r2_1_UNREAL	Yes*	No	-	-
gb_s2_r2_2_REAL	No	No	No	No
gb_s2_r2_3_REAL	No	No	No	No
gb_s2_r2_4_REAL	No	No	No	No
moving_obstacle_24x24_7glitches	Yes	Yes	No	No
moving_obstacle_32x32_11glitches	Yes	Yes	No	No
moving_obstacle_48x48_19glitches	Yes	Yes	No	No
moving_obstacle_64x64_27glitches	Yes	Yes	No	No
moving_obstacle_96x96_43glitches	Yes	Yes	No	No
moving_obstacle_128x128_59glitches	No	No	-	-
moving_obstacle_128x128_60glitches	No	No	-	-
mult11	Yes	Yes*	-	-
mult12	Yes*	No	No	No
mult13	Yes*	No	-	-
mult14	Yes*	No	-	-
mult15	Yes*	No	-	-
mult16	Yes*	No	No	No
stay16y	Yes	Yes*	Yes	Yes*
stay18n	Yes*	No	-	-
stay18y	No	No	-	-
stay20n	Yes*	No	-	-
stay20y	No	No	-	-
stay22n	No	No	-	-
stay22y	No	No	-	-
stay24n	No	No	-	-
stay24y	No	No	-	-

solved within the timeout of 5000 seconds can actually be solved very quickly, i.e., in less than 600 seconds.

The uniquely solved benchmarks also show that there are significant differences between the algorithms of different tools. In particular, the template-based variant of Demiurge, while not very successful overall, can determine realizability for a relatively large number of Toy Examples that cannot be solved by the other approaches.

Regarding the relative ranking, in Table 2 we note that Demiurge (learn) has the same score as Basil (and higher than AbsSynthe), even though it can only solve 359 problems, compared to 520 for Basil and 536 for AbsSynthe. This is because this ranking rewards Demiurge for being one of the fastest tools on many of the small problem instances.

Parallel Subtrack. The parallel realizability subtrack had 4 participants: parallel versions of Demiurge and Re-alizer[31], and sequential versions of AbsSynthe and Basil. The results are summarized in Table 4. Again, no tool could solve all 569 benchmarks. Table 3 shows 21 benchmarks that were not solved by any of the tools within the timeout, and 6 benchmarks that were solved uniquely by one tool. The successful tools are:

- Basil: 4 versions of the factory assembly benchmarks (the same as before)
- AbsSynthe: mult11 and stay16y

We note that Realizer could not solve the problems that it uniquely solved in sequential execution mode. Furthermore, Demiurge (templ) did not compete in the parallel subtrack, therefore its uniquely solved problems from the sequential subtrack are unsolved here.

---

[31] Due to a bug, the parallel version of Realizer performed worse than the sequential version, as mentioned in Section 6.

Table 4. Results of the Parallel Realizability Track

Tool	Solved	Unique	Relative
Realizer	**538**	0	279
AbsSynthe	536	2	219
Basil	520	**4**	**331**
Demiurge (parallel)	324	0	133

The best result in each column is in bold.

Table 5. Results of the Sequential Synthesis Track

Tool	Solved	Relative	Quality	MCTO
AbsSynthe	**143**	329	**265**	6
Demiurge (learn)	121	**379**	240	0
Basil	117	218	219	5
Demiurge (templ)	31	77	57	0

The best result in each column is in bold.

A cactus plot for the number of benchmarks that can be solved by each tool within the timeout is given in Figure 4. We do not give a detailed analysis of the number of solved instances by category, since it is very similar to the analysis in Figure 2.

Analysis. Figure 4 shows the same steep degradation of runtime with increasing complexity as in the sequential case. Concerning the effectiveness of parallel versus sequential implementations, this subtrack shows that currently none of the implementations benefits from using parallelism. To the contrary, the parallel implementations of Demiurge (learn) and Realizer were not able to solve (in 5000s Wall time) the problems that their sequential implementations solved uniquely in the sequential subtrack (in 5000s CPU time), and AbsSynthe and Basil are sequential implementations.

Regarding the relative ranking based on Wall time, we note another weakness of the chosen ranking system: this ranking heavily favors implementations in C++ that have a quick startup time. Basil solves 150 problems in less than 0.36s Wall time, which is the minimal time needed to solve any single problem for the Python-based implementations AbsSynthe and Realizer. That is, the relative ranking scheme and our benchmark selection favors tools that can solve easy benchmarks very quickly, and in particular the tools implemented in C++, as they make very efficient use of Wall time.

Summing up, we see that in the realizability tracks the BDD-based tools in general outperform the SAT- and QBF-based approaches, except for a small subset of the benchmarks. Between the BDD-based approaches, the differences we could detect are rather small — the percentage of benchmarks that can be solved by the BDD-based implementations ranges only from 91% to 95%. None of the tools benefits from parallelism.

8.2 Synthesis Track

In the synthesis track, tools were evaluated with respect to the relative and quality rankings (see Section 5.2), based on the size of solutions. Since these rankings are only defined on realizable specifications, we excluded all unrealizable specifications. Furthermore, we excluded most of the problems that could not be solved by any tool in the realizability track, since synthesis in general is harder than realizability checking. Out of the remaining 382 benchmarks, we chose 157 benchmarks with the goal to ensure a good coverage of different benchmark classes, and a good distribution over benchmarks of different difficulty. Only 3 out of the 5 tools that entered SYNTCOMP competed in the synthesis track: AbsSynthe, Basil and Demiurge.

Sequential Subtrack. The sequential synthesis subtrack had 4 participants: AbsSynthe and Basil competed with one configuration each, and Demiurge with two different configurations. Table 5 shows the number of solved instances, and the accumulated points per tool in the relative and quality rankings. Note that a problem only counts as solved if the solution is successfully model checked. The number of model checking timeouts (MCTO) is also given in the table.

No tool could solve all 157 benchmarks. No benchmark was solved uniquely by one tool, and 14 benchmarks were solved by none of the tools (see Table 3). AbsSynthe solved the highest number of problems and earns the highest score in the quality ranking. Demiurge (learn) earns the highest score in our relative ranking, even though it solves less problems than AbsSynthe.

Both AbsSynthe and Basil produced a small number of solutions that could not be model checked within the separate 3600s timeout. While counting these additional solutions would have changed the scores of these tools, it would not have changed the order of tools in any of the rankings. Figures 5 and 6 give an overview of the size of produced implementations for a subset of the benchmarks, showing significant differences on implementation sizes for some instances, in particular from the AMBA and Genbuf classes.

Analysis. Regarding the relative and quality rankings, we note that Demiurge (learn) profits from taking solution sizes into account. Figure 5 shows that for those instances of the AMBA and GenBuf benchmarks that Demiurge (learn) can solve, it provides implementations that are often by an order of magnitude smaller than those of the other tools. Figure 6 shows a number of benchmarks where the implementation sizes are equal or very similar. Most of the time, the solutions of Demiurge (learn) are smaller than those of AbsSynthe and Basil, which is why it scores higher than AbsSynthe and much higher than Basil in the relative ranking, even though it solves less problems than AbsSynthe, and about as many as Basil. In the quality ranking, the relative difference between Demiurge (learn) and AbsSynthe is significantly
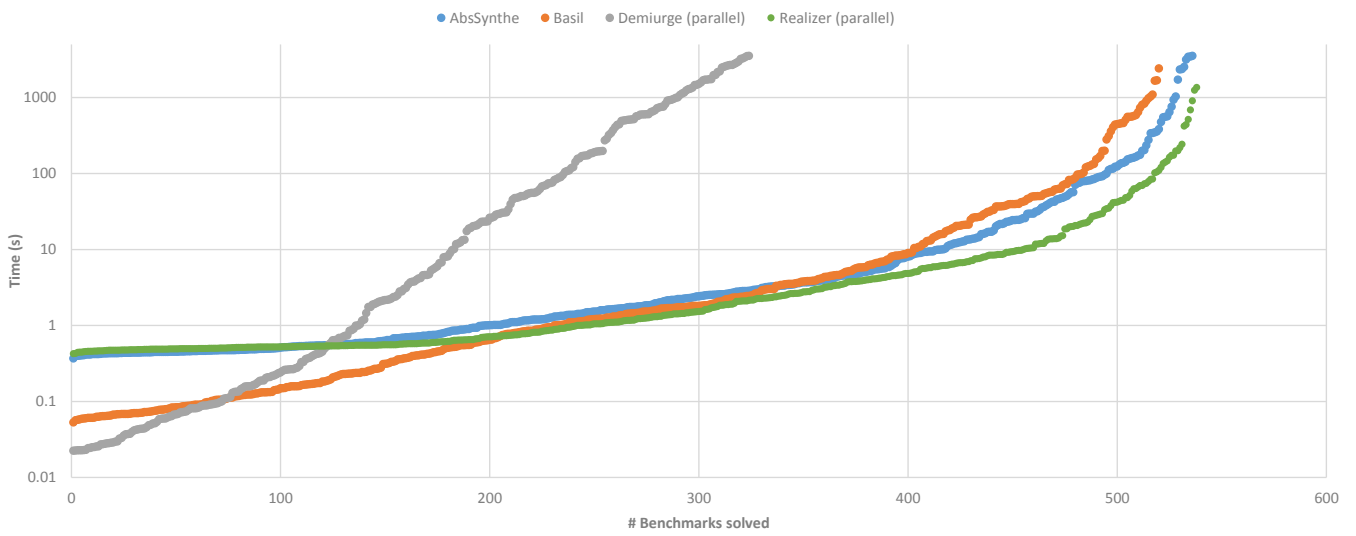
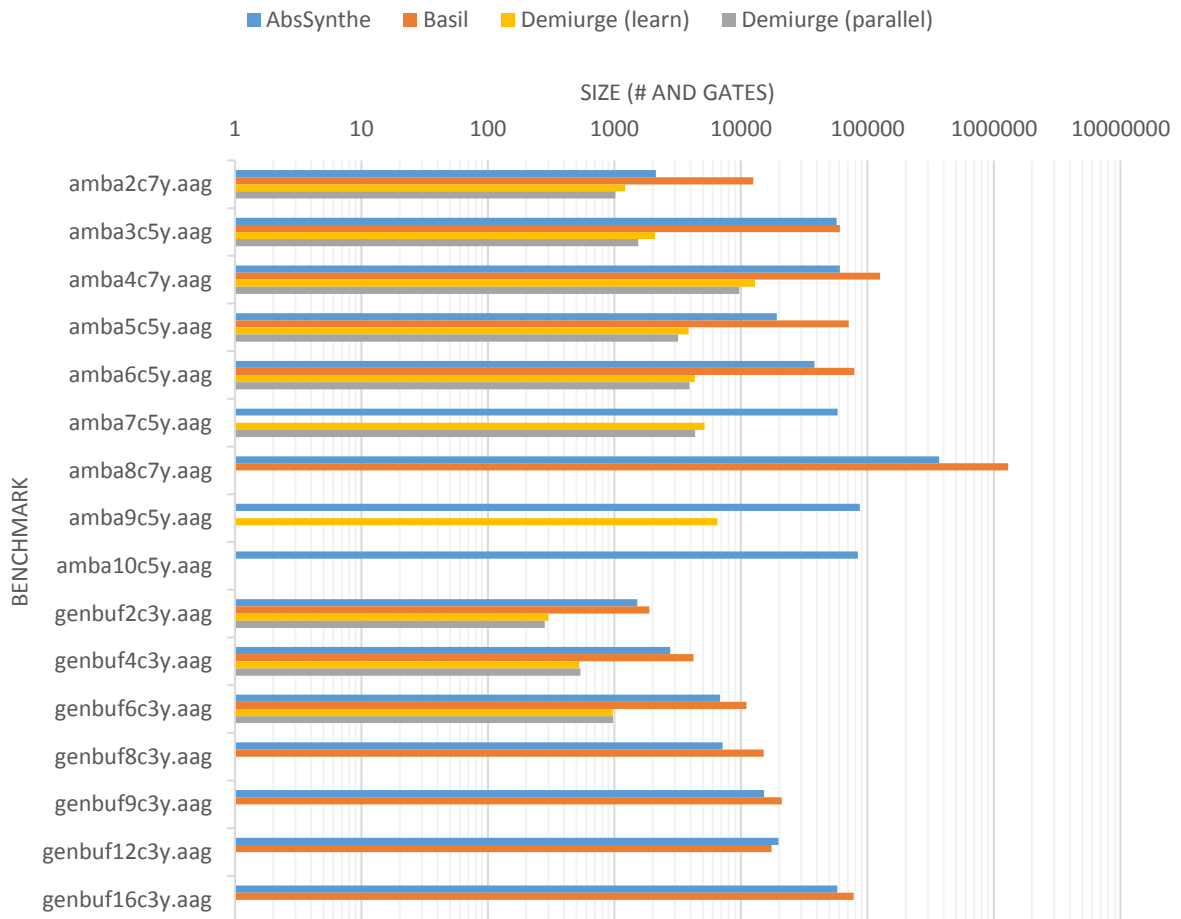**Fig. 4.** Parallel Realizability Track, Cactus Plot

Fig. 5. Comparison of implementation sizes for a subset of the AMBA and GenBuf benchmarks
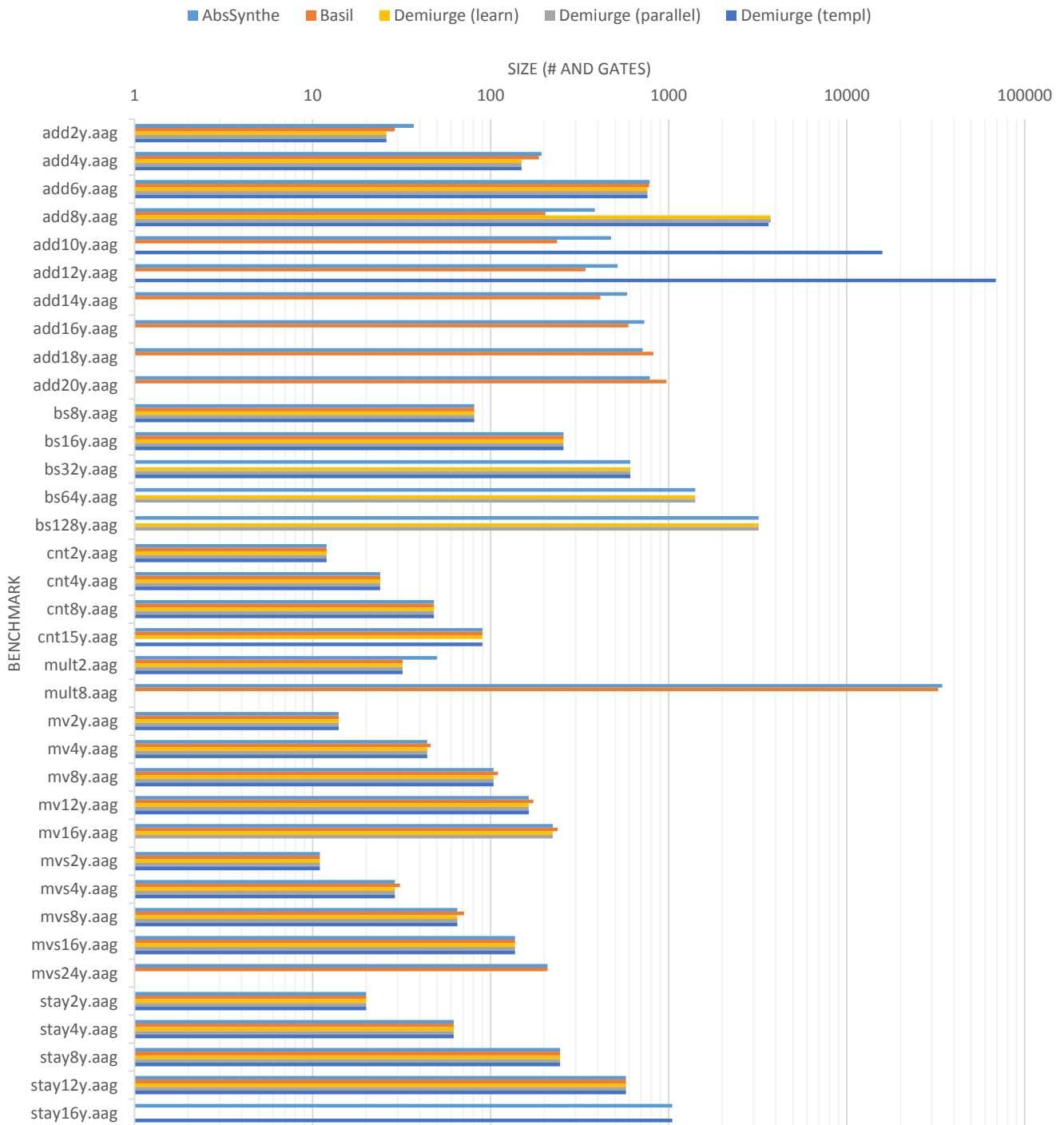
Fig. 6. Comparison of implementation sizes for a subset of the toy example benchmarks

smaller than in the number of solved instances (9.5% versus 15.5% difference).

Furthermore, we note that the benchmark set contains relatively many problems that are easy to solve. For example, AbsSynthe can solve 75 of the 157 problems in less than 0.5s CPU time.

Comparing the BDD-based tools, AbsSynthe solves a number of problems that Basil cannot solve, and provides smaller solutions in many cases.

Parallel Subtrack. The parallel synthesis subtrack had 3 participants: one configuration each of AbsSynthe, Basil, and Demiurge. Demiurge (parallel) was the only tool to use parallelism in the synthesis track. The results are summarized in Table 6. No tool could solve all 157 bench-

Table 6. Results of the Parallel Synthesis Track

Tool	Solved	Relative	Quality	MCTO
AbsSynthe	**143**	352	**266**	6
Demiurge (parallel)	119	**393**	237	0
Basil	117	235	196	5

The best result in each column is in bold.

marks, 3 benchmarks were solved uniquely by one tool, and 14 benchmarks were solved by none of the tools.

The benchmarks solved uniquely by one tool are:

– AbsSynthe: `amba9c5y`, `amba10c5y`, and `stay16y`.

Like in the sequential subtrack, both AbsSynthe and Basil produced a small number of solutions that could not be model checked. Implementation sizes for Demiurge (parallel) are included in Figures 5 and 6, showing that in some cases the implementations are even smaller than those obtained from Demiurge (learn), in particular for the AMBA benchmarks.

Analysis. Like in the sequential synthesis subtrack, Demiurge profits from providing small solutions, even though it solves less problems than its competitors. Furthermore, we note that Demiurge in this case profits from parallelism to some extent. While it solved 2 problems less than the sequential Demiurge (learn), the solutions provided by Demiurge (parallel) were in some cases even smaller than those provided by the sequential version.

8.3 Observations on the State of the Art

BDD-based Synthesis. The standard BDD-based fix-point algorithm for solving safety games is currently the most efficient way for realizability checking based on monitor circuits. Implementations of the algorithm build on existing BDD packages, including operations for composition, abstraction, and dynamic reordering of BDDs. Based on these complex BDD operations, a competitive implementation can be fairly simple, as can be seen for example in Simple BDD Solver, which only consists of about 300 lines of code. A few optimizations seem to be crucial, like automatic reordering, partitioned transition relations, and direct substitution. For other optimizations, like eager deallocation of BDDs or simultaneous conjunction and abstraction, we have mixed results: the tool authors that implemented them report increased efficiency, but we also have competitive tools that do not implement them.

A drawback of BDD-based synthesis becomes apparent when comparing the size of solutions to those of Demiurge (learn): in many cases, the produced implementations are much larger than necessary.

As can be expected, a deeper analysis of the runtime behavior of BDD-based tools shows that most of the time is spent manipulating BDDs, in particular in the automatic reordering operations. Therefore, it can be expected that the performance of BDD-based implementations heavily depends on the performance of the used BDD package. Since all of the tools in SYNTCOMP 2014 use the same BDD package, the results of the competition do not shed light on this issue, however.

Template-based Synthesis. The template-based algorithm implemented in Demiurge (templ) only solves a small subset of the benchmark set — a closer analysis shows that it only performs well if a simple CNF representation of the winning region exists, which applies only to few SYNTCOMP benchmarks. Hence, its performance on average is rather poor. However, this approach solves large instances of the mult, cnt and stay benchmarks much faster than the competition, or solves them uniquely.

Learning-based Synthesis. The learning-based algorithm implemented in Demiurge (learn) solves far more benchmarks than the template-based algorithm: 62% of the benchmarks instead of 21% in the sequential realizability track. Still, the approach cannot really compete with the BDD-based tools, which solve more than 90%. In the parallel realizability track, the situation is similar.

In the synthesis tracks, which are restricted to realizable problems and have rankings that take into account the size of solutions, Demiurge (learn) performed much better. Here, it solves 77% of the benchmarks, compared to 78% for Basil and 95% for AbsSynthe (before model checking). Additionally, the learning-based algorithm produces circuits that are sometimes several orders of magnitude smaller than those produced by the BDD-based tools. This is also highlighted by the fact that all solutions of Demiurge (learn) are successfully model checked, while both AbsSynthe and Basil produce a number of solutions that can not be verified within the timeout.

Parallel Subtracks. The submitted tools in general do not use parallelization very efficiently. The parallel version of Realizer performs worse than the sequential version due to a bug. For the parallel version of Demiurge, the result is double-edged: on the one hand, the parallel version solves 2 problems less than the sequential version, on the other hand the solutions provided are often even smaller than the ones produced by the sequential version.

For BDD-based tools, the lack of efficient parallel implementations correlates with the lack of efficient parallelized operations in BDD packages. While there have been recent efforts to parallelize BDD operations [64,65], this package does not support the important automatic reordering of BDDs, which makes it hard to integrate into a technique that heavily relies on reordering.

9 Conclusions and Future Plans

SYNTCOMP 2014 was a big success, making the first step towards establishing the competition as a regular event and its benchmark format as a standard language in the synthesis community. A number of synthesis tools have been developed specifically for the competition (AbsSynthe, Basil, Realizer), while others are new versions or modifications of existing tools (Demiurge, Simple BDD Solver). Recently, the competition format has also been adopted by tool developers that have thus far not participated in SYNTCOMP [22]. Furthermore, the competition has sparked a lively discussion on the implementation of efficient synthesis techniques, in particular making tool developers aware of the range of optimizations used in BDD-based synthesis algorithms, and alternative SAT- and QBF-based approaches that are competitive at least on some classes of benchmarks.

At the time of this writing, SYNTCOMP 2015 has already been held [41]. For the second iteration of the competition, we have expanded the benchmark set to more challenging benchmarks, and to a wider range of different benchmark classes. Additionally, following ideas of Sutcliffe and Suttner [61] we have developed a classification scheme for benchmarks in terms of difficulty, based on the results of SYNTCOMP 2014. Using this classification, in SYNTCOMP 2015 we selected benchmarks to balance the weight of benchmark instances from different classes and different difficulties.

Finally, recall that SYNTCOMP 2014 (and 2015) was restricted to the synthesis of finite-state systems from pure safety specifications in AIGER format. On the one hand, this resulted in a low entry-barrier for the competition and revived interest in the synthesis from pure safety specifications, as witnessed by several new tools and research papers related to the competition [9,13,18]. On the other hand, many of the existing synthesis tools did not participate because their strengths are in different kinds of synthesis tasks, for example in the synthesis from specifications in richer specification languages such as GR(1) or LTL. Thus, many interesting synthesis approaches are currently not covered by the competition. For SYNTCOMP 2016, we plan to extend the competition to a specification format that includes both GR(1) and LTL specifications [42].

References

1. R. Alur, R. Bodík, E. Dallal, D. Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madhusudan, M. M. K. Martin, M. Raghothaman, S. Saha, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Dependable Software Systems Engineering*, volume 40 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 1–25. IOS Press, 2015.

2. R. Alur, P. Madhusudan, and W. Nam. Symbolic computational techniques for solving games. *STTT*, 7(2):118–128, 2005.

3. A. Aziz, S. Tasiran, and R. K. Brayton. BDD variable ordering for interacting finite state machines. In *DAC*, pages 283–288, 1994.

4. A. Balint, D. Diepold, D. Gall, S. Gerber, G. Kapler, and R. Retz. EDACC - an advanced platform for the experiment design, administration and analysis of empirical algorithms. In *LION 5. Selected Papers*, volume 6683 of *LNCS*, pages 586–599. Springer, 2011.

5. C. W. Barrett, L. M. de Moura, and A. Stump. Design and results of the first satisfiability modulo theories competition (SMT-COMP 2005). *J. Autom. Reasoning*, 35(4):373–390, 2005.

6. D. Beyer. Competition on software verification - (SV-COMP). In *TACAS*, volume 7214 of *LNCS*, pages 504–524. Springer, 2012.

7. D. Beyer, S. Löwe, and P. Wendler. Benchmarking and resource measurement. In *SPIN 2015*, volume 9232 of *LNCS*, pages 160–178. Springer, 2015.

8. R. Bloem, A. Cimatti, K. Greimel, G. Hofferek, R. Könighofer, M. Roveri, V. Schuppan, and R. Seeber. RATSY - A new requirements analysis tool with synthesis. In *CAV*, volume 6174 of *LNCS*, pages 425–429. Springer, 2010.

9. R. Bloem, U. Egly, P. Klampfl, R. Könighofer, and F. Lonsing. SAT-based methods for circuit synthesis. In *FMCAD'14*, pages 31–34. IEEE, 2014.

10. R. Bloem, S. J. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Automatic hardware synthesis from specifications: a case study. In *DATE*, pages 1188–1193. ACM, 2007.

11. R. Bloem, S. J. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Specify, compile, run: Hardware from PSL. *Electr. Notes Theor. Comput. Sci.*, 190(4):3–16, 2007.

12. R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. *J. Comput. Syst. Sci.*, 78(3):911–938, 2012.

13. R. Bloem, R. Könighofer, and M. Seidl. SAT-based synthesis methods for safety specs. In *VMCAI*, volume 8318 of *LNCS*, pages 1–20. Springer, 2014.

14. A. Bohy, V. Bruyère, E. Filiot, N. Jin, and J.-F. Raskin. Acacia+, a tool for LTL synthesis. In *CAV*, volume 7358 of *LNCS*, pages 652–657. Springer, 2012.

15. A. R. Bradley. SAT-based model checking without unrolling. In *VMCAI*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011.

16. R. K. Brayton, G. D. Hachtel, A. L. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. A. Edwards, S. P. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: A system for verification and synthesis. In *CAV*, volume 1102 of *LNCS*, pages 428–432. Springer, 1996.

17. R. K. Brayton and A. Mishchenko. ABC: An academic industrial-strength verification tool. In *CAV*, volume 6174 of *LNCS*, pages 24–40. Springer, 2010.

18. R. Brenguier, G. A. Pérez, J.-F. Raskin, and O. Sankur. AbsSynthe: abstract synthesis from succinct safety specifications. In *SYNT*, volume 157 of *EPTCS*, pages 100–116. Open Publishing Association, 2014.

19. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

20. J. Büchi and L. Landweber. Solving sequential conditions by finite-state strategies. *Trans. Amer. Math. Soc.*, 138:295–311, 1969.

21. J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transistion relations. In *VLSI*, pages 49–58, 1991.

22. T. Chiang and J. R. Jiang. Property-directed synthesis of reactive systems from safety specifications. In *ICCAD*, pages 794–801. ACM, 2015.

23. A. Church. Logic, arithmetic and automata. In *Proceedings of the international congress of mathematicians*, pages 23–35, 1962.

24. O. Coudert and J. C. Madre. A unified framework for the formal verification of sequential circuits. In *ICCAD*, pages 126–129, 1990.

25. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

26. L. de Alfaro and P. Roy. Solving games via three-valued abstraction refinement. In *CONCUR*, volume 4703 of *LNCS*, pages 74–89. Springer, 2007.

27. R. Ehlers. Experimental aspects of synthesis. In *iWIGP*, volume 50 of *EPTCS*, pages 1–16, 2011.

28. R. Ehlers. Unbeast: Symbolic bounded synthesis. In *TACAS*, volume 6605 of *LNCS*, pages 272–275. Springer, 2011.

29. R. Ehlers. Symbolic bounded synthesis. *Formal Methods in System Design*, 40(2):232–262, 2012.

30. R. Ehlers, R. Könighofer, and G. Hofferek. Symbolically synthesizing small circuits. In *FMCAD'12*, pages 91–100. IEEE, 2012.

31. E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.*, 2(3):241–266, 1982.

32. E. Filiot, N. Jin, and J. Raskin. Exploiting structure in LTL synthesis. *STTT*, 15(5-6):541–561, 2013.

33. E. Filiot, N. Jin, and J.-F. Raskin. Antichains and compositional algorithms for LTL synthesis. *Formal Methods in System Design*, 39(3):261–296, 2011.

34. B. Finkbeiner and S. Jacobs. Lazy synthesis. In *VMCAI*, volume 7148 of *LNCS*, pages 219–234. Springer, 2012.

35. B. Finkbeiner and S. Schewe. Bounded synthesis. *STTT*, 15(5-6):519–539, 2013.

36. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

37. T. A. Henzinger, R. Jhala, and R. Majumdar. Counterexample-guided control. In *ICALP*, volume 2719 of *LNCS*, pages 886–902. Springer, 2003.

38. Y. Hong, P. A. Beerel, J. R. Burch, and K. L. McMillan. Sibling-substitution-based BDD minimization using don't cares. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 19(1):44–55, 2000.

39. S. Jacobs. Extended AIGER format for synthesis. *CoRR*, abs/1405.5793, 2014.

40. S. Jacobs, R. Bloem, R. Brenguier, R. Ehlers, T. Hell, R. Könighofer, G. A. Pérez, J. Raskin, L. Ryzhyk, O. Sankur, M. Seidl, L. Tentrup, and A. Walker. The first reactive synthesis competition (SYNTCOMP 2014). *CoRR*, abs/1506.08726, 2015.

41. S. Jacobs, R. Bloem, R. Brenguier, R. Könighofer, G. A. Pérez, J.-F. Raskin, L. Ryzhyk, O. Sankur, M. Seidl, L. Tentrup, and A. Walker. The second reactive synthesis competition (SYNTCOMP 2015). In *SYNT*, volume 202 of *EPTCS*, pages 27–57. Open Publishing Association, 2016.

42. S. Jacobs and F. Klein. A high-level LTL synthesis format: TLSF v1.0. *CoRR*, abs/1601.05228, 2016.

43. M. Järvisalo, D. L. Berre, O. Roussel, and L. Simon. The international SAT solver competitions. *AI Magazine*, 33(1), 2012.

44. B. Jobstmann and R. Bloem. Optimizations for LTL synthesis. In *FMCAD*, pages 117–124. IEEE Computer Society, 2006.

45. B. Jobstmann, S. J. Galler, M. Weiglhofer, and R. Bloem. Anzu: A tool for property synthesis. In *CAV*, volume 4590 of *LNCS*, pages 258–262. Springer, 2007.

46. O. Kupferman and M. Y. Vardi. Safraless decision procedures. In *FOCS*, pages 531–542. IEEE Computer Society, 2005.

47. R. P. Kurshan. Automata-theoretic verification of coordinating processes. In *Analysis and Optimization of Systems: Discrete Event Systems*, pages 16–28. Springer, 1994.

48. C. Lecoutre, O. Roussel, and M. R. C. van Dongen. Promoting robust black-box solvers through competitions. *Constraints*, 15(3):317–326, 2010.

49. A. Mishchenko, S. Chatterjee, and R. K. Brayton. Dag-aware AIG rewriting a fresh look at combinational logic synthesis. In *DAC*, pages 532–535. ACM, 2006.

50. A. Mishchenko, S. Chatterjee, R. Jiang, and R. Brayton. FRAIGs: A unifying representation for logic synthesis and verification. Technical report, EECS Dept., U. C. Berkeley, 2005.

51. A. Morgenstern, M. Gesell, and K. Schneider. Solving games using incremental induction. In *IFM'13*, LNCS 7940, pages 177–191. Springer, 2013.

52. A. Niemetz, M. Preiner, F. Lonsing, M. Seidl, and A. Biere. Resolution-based certificate extraction for QBF. In *SAT'12*, LNCS 7317, pages 430–435. Springer, 2012.

53. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL*, pages 179–190. ACM Press, 1989.

54. M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Trans. Amer. Math. Soc.*, 141:1–35, 1969.

55. R. K. Ranjan, A. Aziz, R. K. Brayton, B. Plessier, and C. Pixley. Efficient bdd algorithms for fsm synthesis and verification. In *International Workshop on Logic Synthesis*, 1995.

56. O. Roussel. Controlling a solver execution with the runsolver tool. *JSAT*, 7(4):139–144, 2011.

57. R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD*, pages 42–47. IEEE Computer Society, 1993.

58. M. Seidl and R. Könighofer. Partial witnesses from preprocessed quantified boolean formulas. In *DATE'14*, pages 1–6. IEEE, 2014.

59. S. Sohail and F. Somenzi. Safety first: a two-stage algorithm for the synthesis of reactive systems. *STTT*, 15(5-6):433–454, 2013.

60. F. Somenzi. Binary decision diagrams. In *Calculational system design*, volume 173, page 303. IOS Press, 1999.

61. G. Sutcliffe and C. B. Suttner. Evaluating general purpose automated theorem proving systems. *Artif. Intell.*, 131(1-2):39–54, 2001.

62. G. Sutcliffe and C. B. Suttner. The state of CASC. *AI Commun.*, 19(1):35–48, 2006.

63. W. Thomas. On the synthesis of strategies in infinite games. In *STACS*, pages 1–13, 1995.

64. T. van Dijk, A. Laarman, and J. van de Pol. Multi-core BDD operations for symbolic reachability. *Electr. Notes Theor. Comput. Sci.*, 296:127–143, 2013.

65. T. van Dijk and J. van de Pol. Sylvan: Multi-core decision diagrams. In *TACAS 2015*, volume 9035 of *LNCS*, pages 677–691. Springer, 2015.