The Second Reactive Synthesis Competition (SYNTCOMP 2015)

Swen Jacobs
Saarland University
Saarbrücken, Germany

Roderick Bloem
Graz University of Technology
Graz, Austria

Romain Brenguier
Université Libre de Bruxelles
Brussels, Belgium

Robert Könighofer
Graz University of Technology
Graz, Austria

Guillermo A. Pérez
Université Libre de Bruxelles
Brussels, Belgium

Jean-François Raskin
Université Libre de Bruxelles
Brussels, Belgium

Leonid Ryzhyk
NICTA
Sydney, Australia
Carnegie Mellon University
Pittsburgh, USA

Ocan Sankur
Université Libre de Bruxelles
Brussels, Belgium

Martina Seidl
Johannes-Kepler-University
Linz, Austria

Leander Tentrup
Saarland University
Saarbrücken, Germany

Adam Walker
NICTA
Sydney, Australia

We report on the design and results of the second reactive synthesis competition (SYNTCOMP 2015). We describe our extended benchmark library, with 6 completely new sets of benchmarks, and additional challenging instances for 4 of the benchmark sets that were already used in SYNTCOMP 2014. To enhance the analysis of experimental results, we introduce an extension of our benchmark format with meta-information, including a difficulty rating and a reference size for solutions. Tools are evaluated on a set of 250 benchmarks, selected to provide a good coverage of benchmarks from all classes and difficulties. We report on changes of the evaluation scheme and the experimental setup. Finally, we describe the entrants into SYNTCOMP 2015, as well as the results of our experimental evaluation. In our analysis, we emphasize progress over the tools that participated last year.

1 Introduction

The automatic synthesis of reactive systems from formal specifications is one of the major challenges of computer science. While there has been lively research on the topic for more than 50 years and a number of fundamental approaches to solve the problem have been proposed [23, 25, 40, 41], the impact of theoretical results on the practice of system design has been rather limited. This is in spite of the obvious advantages of the automatic construction of provably correct systems and an increased interest in possible applications of reactive synthesis techniques, e.g., in robotics and cyber-physical systems. [21, 22, 34] The goal of the reactive synthesis competition (SYNTCOMP) is to increase the impact of theoretical advancements in synthesis by fostering research in scalable and user-friendly implementations of synthesis techniques.

In particular, SYNTCOMP aims at

i) making synthesis tools comparable by establishing a standard benchmark format,

ii) facilitating the exchange of benchmarks by providing a public benchmark repository and encouraging researchers to contribute their benchmarks,

iii) enabling a comprehensive and fair evaluation of synthesis tools by providing a *dedicated and independent* platform for the comparison of tools under consistent experimental conditions,

iv) encouraging the implementation of synthesis tools that can be used as *black-box solvers* in applications by enforcing that competition entrants have to run on the complete benchmark set of SYNTCOMP with a fixed configuration, and

v) fostering the efficient implementation of synthesis algorithms by regularly providing new and challenging benchmark problems, and comparing the performance of tools on these.

The first SYNTCOMP was held during June and July of 2014, and its results were first presented at the 26th International Conference on Computer Aided Verification (CAV) and the 3rd Workshop on Synthesis (SYNT) in July 2014. [30] A design choice of the first competition was to focus on safety properties specified as monitor circuits in an extension of the AIGER format (known from the hardware model checking competition [7]). We keep this restriction for the second competition, and plan to consider extensions of the format in the future.

The organization team of SYNTCOMP 2015 consisted of Roderick Bloem and Swen Jacobs. Based on results and experiences from the first competition, the specific goals for the second competition were:

- to expand the benchmark library with new and challenging problems,
- to improve the selection of benchmarks to ensure an even distribution over benchmarks from different classes and difficulties,
- to improve the ranking scheme, focusing on the most important properties of synthesis tools, and
- to determine the progress of the state of the art by comparing the performance of new entrants to those from last year.

The rest of this paper describes the design, benchmarks, participants, and results of SYNTCOMP 2015. We describe the synthesis problem as considered in SYNTCOMP in Section 2, followed by a presentation of the benchmark set for SYNTCOMP 2015 in Section 3. Section 4 introduces the rules and setup of the competition, including classification and selection of benchmarks. In Section 5 we give an overview of the entrants of SYNTCOMP 2015, focusing on changes with respect to last year's entrants. Finally, we present and analyze the experimental results in Section 6.

Note that Sections 3 and 5 are in part based on descriptions supplied by the respective authors of benchmarks and tools. The remainder of this article is original work of the SYNTCOMP organizers.

2   Synthesis Problem

We briefly summarize the reactive synthesis problem as it is considered in SYNTCOMP. A more detailed introduction into the problem, as well as into the different approaches for solving it, can be found in [30].

In SYNTCOMP, we consider the automatic synthesis of finite-state reactive systems that satisfy a safety specification, given as a monitor circuit that raises a special output err when an unsafe state is visited. The specification is encoded in the SYNTCOMP format, an extension of the well-known AIGER format [6] that allows inputs of the circuit to be defined as either controllable or uncontrollable. In the traditional game-based approach to the synthesis of reactive systems [20, 41, 47], such a specification gives rise to a *game* between two players: states of the game are given by the valuation of latches in the monitor circuit, the environment player decides on the uncontrollable inputs of the specification circuit, and the system player decides on the controllable inputs. The goal of the system player is to satisfy the specification, i.e., to visit only safe states, independent of the environment behavior.

Algorithms that solve this game usually take an approach that consists of two steps. In the first step, a so-called *winning region* is computed. The winning region W is the set of all states from which the system player can enforce to satisfy the specification, i.e., to visit only safe states in the subsequent

computation. In the classical algorithm, this is done by computing the fixpoint of the *uncontrollable predecessor* operation `upre` on the error states, i.e., inductively computing all 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 of the system player. In a second step, a *winning strategy* is derived from the winning region. For every (current) state and uncontrollable input, the winning strategy defines a set of controllable inputs that are okay for satisfying the specification. In order to obtain an implementation of this strategy as a circuit, a concrete choice for the controllable inputs has to be made for every state and uncontrollable input.

In order to achieve acceptable scalability, it is important to implement synthesis algorithms symbolically, i.e., by manipulating formulas instead of enumerating states. In synthesis, these symbolic algorithms are usually implemented with Binary Decision Diagrams (BDDs) [4, 19, 46]. One reason for this is that solving games inherently involves dealing with quantifier alternations, and BDDs offer techniques for handling both kinds of quantification. However, BDDs also have their scalability issues. On the other hand, there have been enormous performance improvements in decision procedures for the satisfiability of (boolean) formulas over the last years and decades. This has lead to efficient tools like SAT- and QBF (Quantified Boolean Formulas) solvers, which can also be leveraged to obtain efficient symbolic synthesis algorithms.

All of the tools that compete in SYNTCOMP 2015 implement symbolic game-based synthesis in some form. A description of the tools will be given in Section 5.

3 Benchmarks

In this section, we describe the benchmark library for SYNTCOMP 2015. We start by describing in detail 6 new classes of benchmarks. This is followed by short descriptions of the classes of benchmarks that have already been used in SYNTCOMP 2014 — for some of these, additional problem instances have been added, and two of them are split into multiple classes for SYNTCOMP 2015.

3.1 Scheduling of Washing Cycles (new class)

This class of benchmarks specifies a washing system, with cycles that can be launched in parallel. The system is composed of several tanks that may depend on shared water pipes, and can be controlled by the user pushing buttons. It is parameterized in the number of tanks n, the maximum allowed reaction delay d, as well as the number t of tanks that share a water pipe. The correct behavior of the system is described by rational expressions that impose safety constraints. These safety constraints are first translated to non-deterministic automata, and then into the SYNTCOMP format. The benchmark set is described in more detail in [18].

Problem description. We consider the design of a centralized controller for a washing system, composed of n tanks running in parallel. The user can request the delivery of water to a given tank i by pressing a button, modeled as uncontrollable input $push_i$. The release of water into the tank is modeled by controllable input $fill_i$. After a determined time k, the controller should open the valve to remove the water from inside the tank, modeled by controllable input $empty_i$. Finally, there is a controllable input light that models an activation light that should be on whenever one of the tanks contain water. This is illustrated in Fig. 1.
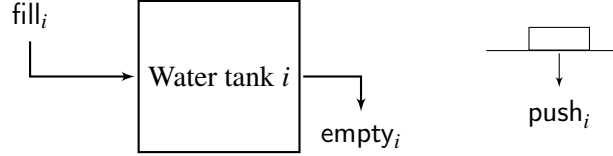
Figure 1: A water tank. Input fill_i controls the arrival of water into the tank. Input empty_i controls the departure of water out of the tank. Input push_i initiates the washing cycle when the button is pushed.

Encoding of the problem by safety specification.   We encode the specification for the controllers into rational expressions that denote *the occurrence of an error* in the execution:

- An error occurs if a button was pressed but the tank is not filled after delay $d \in \mathbb{N}$:

$$A_i = \text{true}^* \cdot \{\text{push}_i\} \cdot \{\neg\text{fill}_i\}^d.$$

- The tank should not be filled unless the button was pushed:

$$B_i = \text{true}^* \cdot \{\neg\text{push}_i\}^d \cdot \{\text{fill}_i\}.$$

- The tank should be emptied after a delay of exactly k steps:

$$C_i = \text{true}^* \cdot \{\text{fill}_i\} \cdot \text{true}^k \cdot \{\neg\text{empty}_i\}$$
$$C_i' = \text{true}^* \cdot \{\text{fill}_i\} \cdot (\text{true} \mid \varepsilon)^{k-1} \{\text{empty}_i\}$$

- The light should be on if, and only if, one of the n tanks is being filled:

$$D = \text{true}^* \cdot \left\{ \text{light} \neq \bigvee_{i \in [1,n]} \text{fill}_i \right\}.$$

- If tanks i and j are connected to the same pipe, as illustrated in Fig. 2, then the system should not activate fill_i and fill_j at the same time. We write P for the set of pipes, and for a pipe $p \in P$, we write $i \in p$ to denote that tank i is fed by pipe p. We encode the mutual exclusion constraint by raising an error for the following expression:

$$E = \text{true}^* \cdot \left\{ \bigvee_{p \in P} \bigwedge_{i,j \in p} \text{fill}_i \right\}$$

Overall, we consider the safety specification given by the language:

$$\mathscr{L} = \bigcup_{i \in [1,n]} \left(L(A_i) \cup L(B_i) \cup L(C_i) \cup L(C_i') \right) \cup L(D) \cup L(E)$$

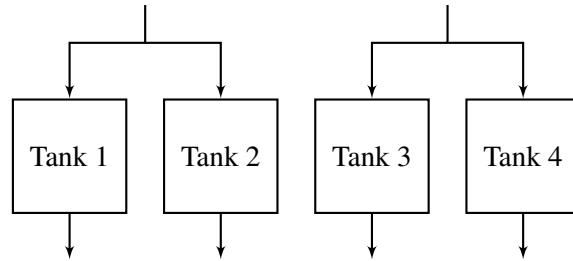Where $L(A)$ denotes the languages associated to expresssion A.

Figure 2: A system with 4 water tanks. Tanks 1 and 2 connect to the same pipe, and so do tanks 3 and 4.

Translation to AIGER.   To produce benchmarks in AIGER format, the conjunction of rational expressions is first translated to a non-deterministic automaton $\mathscr{A}$. Each state of $\mathscr{A}$ is encoded by a latch in the AIGER file. In an execution of the circuit, a latch l is on if, and only if, there is an execution of $\mathscr{A}$ — on the prefix of the word that has been read so far — that leads to the state corresponding to latch l. The error latch is put to true if, and only if, one of the latches corresponding to accepting states is on. The error output will be set to true whenever the current prefix belongs to the language $\mathscr{L}$.

In the benchmark package, the AIGER files are named `cycle_sched_n_d_t.aag`, where n is the number of tanks, d is the maximum delay between $push_i$ and $fill_i$, and t is the number of tanks that are alimented by the same pipes. The delay k between $fill_i$ and $empty_i$ is always chosen to be equal to d.

This benchmark set contains 321 instances, and has been supplied by Romain Brenguier. The OCaml program that has been used to generate AIGER files from rationnal expressions can be downloaded from the following address: `https://github.com/romainbrenguier/Speculoos`.

3.2   Driver Synthesis (new class)

This class of benchmarks is based on a driver synthesis benchmark for the Termite synthesis tool [1, 44]. In Termite, the driver synthesis problem is encoded as a game played by the driver (or controller) against its environment, consisting of an operating system and the device to be controlled. The given benchmark considers the synthesis of a driver for an IDE (or PATA) hard disk controller. The driver must perform three basic functions: read sectors, write sectors and device configuration.

Problem description.   The benchmark consists of two interacting state machines:

- an operating system (OS), and

- an IDE hard disk controller.

The OS acts as a workload generator for generating possible OS-driver interactions. In this benchmark, the specification requires

- that the driver both read and write hard disk sectors,

- that these sectors are eventually read/written, and

- that the driver does not perform erroneous actions such as reading/writing hard disk sectors when not requested.

The requirements that the OS specification imposes are expressed as reachability goals, e.g., any outstanding hard disk write is eventually performed.

The IDE hard disk controller state machine models the external register-mapped control interface of this device as well as an internal state machine that performs the disk sector reads and writes. By writing the correct values to these registers in the correct order, the driver can initiate read and write transactions in order to satisfy the requirements imposed by the OS workload generator. Most registers contain data such as the sector offset to start writing, number of sectors to write, and a pointer to the data's location in memory. Others contain control bits, which, when written, trigger various operations in the hardware such as a disk write.

The goal is to synthesize a driver that provides values for the control bits such that the specification is satisfied. Conceptually, the driver has full information of the system state and when it detects that there is an outstanding request, it performs the necessary device register reads and writes to force the IDE controller state machine to perform a read or write, thus resolving the request.

Parameterization and encoding into AIGER. The control bits of the hard disk controller are modeled as controllable inputs. The uncontrollable inputs model non-determinism in the OS and device state machines. This ensures that the driver can respond correctly to all possible generated workloads.

Benchmarks from this class are named `driver_XYZ.aag`, where X, Y and Z stand for the three dimensions in which the benchmark is parameterized:

- the specification contains a number of state variables that represent data that is inconsequential to the property to be synthesized. One dimension of parameterization is an abstraction of such state variables, by either removing them completely, or reducing the bit-width of certain signals. X=a stands for the variant of the benchmark without data abstraction, and X=b,c,d are three variants that abstract increasingly more of the data.

- the original specification contains a liveness constraint, which is reduced to a safety property by requiring the desired property to hold at least once within every execution fragment of some fixed length, given by Y. Y ranges between 2 and 10, where small values result in unrealizability of the benchmark, while large values increase the size of the state space and potentially make it more difficult to find a solution.

- finally, Z can take two values, where Z=n means that the benchmark has not been further modified, and Z=y means that ABC [16] has been used to simplify the circuit.

This benchmark was supplied by Adam Walker, and translated from Verilog into AIGER by Robert Könighofer, using the VL2MV routine of the VIS system [15], followed by a translation to AIGER format (and possibly optimization) by ABC. The source code and exact sequence of commands are replicated in the comments section of each benchmark file. Overall, this class contains 72 problem instances.

3.3 Huffman Encoder (new class)

This class of benchmarks is taken from recent work of Khalimov [33] on a framework for specifying synthesis problems. The idea is: given a Huffman decoder [29], synthesize the encoder.

The specification includes a given Huffman decoder, which reads the outputs of the encoder that is to be synthesized, and a monitor circuit that compares environment inputs to the encoder with the outputs of the decoder — they should match. Furthermore, the encoder is required to eventually supply the encoded signals that correspond to its input. The benchmark is parameterized in the safety parameter k that is used to approximate this liveness condition, i.e., we require the encoder to supply the encoded signals within k steps. For the given decoder, the specification is unrealizable if $k \leq 9$, and realizable otherwise.

This benchmark was written in an extended version of the SMV format [36, 37], and converted to AIGER by first translating it into standard, flattened SMV format, and then using `smvtoaig` and a liveness-to-safety approximation to obtain a file in the SYNTCOMP format. For more details on the benchmark, the original specification language and its translation into SYNTCOMP format, we refer to Khalimov [33].

3.4 HWMCC (new class)

This class of benchmarks is based on a subset of the benchmarks from HWMCC 2014 [7]. The idea is to consider verification benchmarks consisting of a system and a safety specification that is not satisfied by the system, and ask the question whether we can synthesize a controller for a given subset of the inputs such that the specification is satisfied.

To obtain the benchmarks in this set, we consider the unsafe instances from the *single safety property* track of HWMCC. Depending on the overall number of inputs of the original benchmark, we consider variants where between 1 and 512 are defined to be controllable. The files retain their original file-names, with `_c0tocx` appended to the filename if inputs 0 to x are declared as controllable. E.g., the benchmark based on `6s210b037.aag`, where 32 inputs have been declared as controllable, is named 6s210b037_c0to31.aag.

This benchmark set contains 110 benchmarks, including some of the largest AIGER files that have been considered in the competition so far, with file size of up to 6MB, corresponding to more than 20000 inputs, 40000 latches, and 200000 AND-gates. The original verification benchmarks have been taken from the HWMCC website [7] and modified by Swen Jacobs.

3.5 HyperLTL (new class)

This class of benchmarks is based on a number of benchmark problems from HyperLTL model checking, as introduced in recent work by Finkbeiner et al. [28]. HyperLTL allows to express properties of multiple executions of the same system, e.g., information-flow properties or symmetry properties. In this case, we consider given implementations of several mutual exclusion protocols inspired by the bakery protocol and the AMBA specification, and symmetry properties like

$$\forall \pi, \pi'. \ (select(0)_\pi \wedge select(1)_{\pi'}) \rightarrow \mathsf{G}(pc(0)_\pi = pc(1)_{\pi'} \wedge pc(1)_\pi = pc(0)_{\pi'},$$

stating that if in an execution π the scheduler has initially selected process 0, and in execution π' it has selected process 1, then over the whole execution the program counters of these processes will be swapped.

The set of benchmarks contains different implementations of bakery protocols and the AMBA protocol, with different symmetry properties.

Encoding. The HyperLTL model checking problems are translated to QBF queries, which are in turn encoded in AIGER format. The idea is that the synthesizer checks the HyperLTL property, and constructs a witness formula (represented as a controller circuit) if this is the case.

The benchmark set contains 21 benchmarks, and has been supplied by Markus Rabe. For more details on HyperLTL model checking and the benchmarks, we refer to Finkbeiner et al. [28].

3.6   Matrix Multiplication (new class)

These benchmarks describe the synthesis of a matrix multiplication circuit. That is, the inputs represent two boolean matrices, and the controller needs to give outputs that represent the product matrix of the inputs (in every step of the system). Matrix multiplication is a basic operation that has many applications in mathematics, physics, and engineering. Implementing this operation with a logical circuit of small size is important to produce hardware at small cost. Furthermore, this is an example of a *compositional* benchmark, as every entry of the output matrix only depends on one column of one input matrix and one row of the other, but not on the rest of the input matrices.

In addition to the basic matrix multiplication benchmarks, the benchmark set includes problems with repeated matrix multiplication, where a boolean matrix is stored in the circuit, and multiplied repeatedly with matrices defined by the current inputs. Half of the inputs are controllable, and the goal is to never obtain a matrix that has a line composed only of 0s or only of 1s.

Encoding.   We consider the set of Booleans $\mathbb{B} = \{0,1\}$ and multiplication of matrices in the Boolean ring $\langle \mathbb{B}, \vee, \wedge, 0, 1 \rangle$. The inputs of the specification circuit encode matrices $A \in \mathbb{B}^{m \times n}$, $B \in \mathbb{B}^{n \times o}$, and $C \in \mathbb{B}^{m \times o}$, with the inputs for matrix C defined as controllable. The error output is set to true if, in any step, $A \cdot B \neq C$. In the benchmark package, the AIGER file `mult_bool_matrix_m_n_o.aag` encodes the mutiplication of a matrix A of dimension $m \times n$ and a matrix B of dimension $n \times o$.

For the variant with repeated matrix multiplication, the inputs of the circuit encode a matrix $A \in \mathbb{B}^{n \times n}$ with $n \in \mathbb{N}$, where inputs corresponding to the first $n/2$ columns are controllable. Furthermore, the circuit stores a matrix $B \in \mathbb{B}^{m \times n}$ with $m \in \mathbb{N}$, initialized to a matrix that alternates between 0 and 1 (i.e., $B_{ij} = 1$ iff $i + j$ is even), and updated in each step to the result of $A \cdot B$. The error output is raised if at any point B is such that either $\exists j. \, \forall i. \, B_{ij} = 0$, or $\exists j. \, \forall i. \, B_{ij} = 1$. In the benchmark package, the AIGER file `mult_bool_matrix_dyn_m_n.aag` encodes the repeated mutiplication of a matrix B of dimension $m \times n$ with an input matrix A of dimension $n \times n$.

This benchmark set was supplied by Romain Brenguier and contains 354 problem instances — 273 for basic matrix multiplication, and 81 for the repeated matrix multiplication variant. The AIGER files for these benchmarks have been generated from an OCaml program that can be downloaded from the following address: `https://github.com/romainbrenguier/Speculoos`.

3.7   AMBA (extended class)

The synthesis of a bus controller for the AMBA specification (see [9]) was already considered as a benchmark last year [30]. The benchmarks have been encoded in Verilog and then translated into AIGER, and are parameterized in three dimensions:

- The number of masters n that want to access the bus.

- The type of liveness-to-safety approximation (three different types).

- A parameter k for the liveness-to-safety approximation, requesting for example that a liveness property has to be satisfied after every k steps of the whole system.

This year, this benchmark set has been extended by considering larger values for n (up to 16) and k (up to 60). Overall, the set now contains 952 instances. This benchmark set has been supplied by Robert Könighofer and is described in more detail in [30].

3.8 Genbuf (extended class)

This benchmark set considers the synthesis of a generalized buffer (as described in [9]), and is parameterized in the same way as the AMBA benchmarks above. This year, we consider values of up to 64 for *n*, and 120 for *k*. This benchmark set now contains 866 instances, supplied by Robert Könighofer, and is described in more detail in [30].

3.9 LTL2AIG (extended class)

This set of benchmarks is based on the benchmark set of the Acacia synthesis tool [12, 26], translated from specifications in LTL using the LTL2AIG tool [30]. In 2014, only a part of the original benchmark set was translated, as the LTL2AIG tool did not support all features of the original specifications. This year, the complete benchmark set was entered into the competition. It now contains:

- 50 instances of demo problems that are based on the test suite of LTL synthesis tool LILY [31], with specifications of traffic lights and arbiters in different complexity,

- 41 instances of ltl2dba and ltl2dpa problems that use the synthesis tool to obtain a deterministic Büchi automaton (dba) or a deterministic parity automaton (dpa) that corresponds to a given LTL formula,

- 42 instances of the gb benchmark, based on the same generalized buffer case study as the benchmark above, but in this case as a direct translation of an LTL specification into SYNTCOMP format, and

- 64 instances of a load balancer benchmark, originally presented with LTL synthesis tool UN-BEAST [24].

This benchmark set has been supplied by Guillermo A. Pérez. The first two sets have already been present last year, the other two have been newly generated based on the new version of the LTL2AIG tool (and replace a small set of instances for the gb and load balancer benchmarks that have been present last year). Instead of a single benchmark class, we now consider this set as 4 different benchmark classes in our selection of benchmarks shown in Table 2.

3.10 Toy Examples (extended class)

This benchmark set considers a number of basic building blocks of circuits, such as an adder (add), a bitshifter (bs), a counter (count), or a multiplier (mult). All of the benchmarks are parameterized in the bitwidth of the input. This year, the benchmark set includes

- 50 instances of benchmark add,

- 18 instances of benchmark bs,

- 28 instances of benchmark cnt,

- 14 instances of benchmark mult,

- 42 instances of benchmarks mv and mvs, and

- 24 instances of benchmark stay.

For benchmarks add and bs, instances with larger bit-width have been added this year. This benchmark set has been supplied by Robert Könighofer and is described in more detail in [30].

3.11 Factory Assembly Line and Moving Obstacle (unchanged)

These two benchmark sets describe a controller for two robot arms on an assembly line, and a moving robot that should avoid a moving obstacle in two-dimensional space, respectively. The factory assembly line benchmark consists of 15 problem instances, and the moving obstacle benchmark of 16. Both sets of benchmarks have been supplied by Rüdiger Ehlers and have already been used in SYNTCOMP 2014 [30].

4 Setup, Rules and Execution

4.1 Classification and Selection of Benchmarks

To facilitate the selection of suitable benchmarks and the evaluation of experimental results, we collected additional information on realizability and difficulty of benchmark problems. For realizable specifications, we additionally determined the smallest known solution, to be stored as a reference size. For benchmarks that were already used in SYNTCOMP 2014, this information was obtained from the experimental results of the previous year. For the new benchmarks, we conducted *classification experiments* by running the three most successful solvers from SYNTCOMP 2014 in sequential realizability mode on a representative subset of the new benchmarks.

Extension of the Benchmark Format.   The data obtained in the classification is included directly in each benchmark file, as a special paragraph of the comments section. This paragraph starts with a line containing only the SYNTCOMP tag "`#!SYNTCOMP`", and ends with a line containing only "`#.`". Between these lines, properties of the benchmark can be defined. The properties defined for SYNTCOMP 2015 are listed in Table 1.

   Both `SOLVED_IN` and `REF_SIZE` may be set to 0 if the problem has not been solved before. An example of a classification paragraph is given in Listing 1.

Selection of Competition Benchmarks.   To ensure a fair and meaningful evaluation of the participants, we want to ensure that benchmarks from different classes of problems have approximately equal weight, and that the competition benchmarks represent a good distribution across different difficulties for each class. To this end, benchmarks are divided into categories that can be either small or large. From each

Table 1: Properties defined for classification of benchmarks in SYNTCOMP 2015

Property	Value
`STATUS`	`realizable`, `unrealizable` or `unknown`
`SOLVED_BY`	the fraction of participants that solved the benchmark in a previous experiment, e.g., 8/8, followed by a description of the experiment in brackets, e.g., `[SYNTCOMP2014-RealSeq]`
`SOLVED_IN`	the minimal time (in seconds) needed to solve the problem in a previous experiment, followed by a description of the experiment in brackets
`REF_SIZE`	the minimal size of a solution (in number of AND gates) produced in a previous experiment

Listing 1: Classification paragraph of benchmark file `add8y.aag`

```
#!SYNTCOMP
STATUS : realizable
SOLVED_BY : 8/8 [SYNTCOMP2014-RealSeq]
SOLVED_IN : 0.008 [SYNTCOMP2014-RealSeq]
REF_SIZE : 203
#.
```

category, we selected a number of problems (usually 16 for large categories, and 8 for small) with an even distribution over difficulties, in terms of the ratio of solvers that were able to solve the benchmark previously, given in the SOLVED_BY tag of the benchmarks. In particular, the selected benchmarks from every set include a fraction of about 20% of benchmarks that have not been solved before. The number of selected problems from each category (cp. Section 3) is given in Table 2.

Table 2: Number of selected Benchmarks per Category

Category	Benchmarks	Category	Benchmarks
AMBA	16	Add (Toy Examples)	8
(Washing) Cycle Scheduling	15	Bitshift (Toy Examples)	8
Demo (LTL2AIG)	16	Count (Toy Examples)	8
Driver Synthesis	16	Genbuf (LTL2AIG)	8
Factory Assembly Line	15	Huffman Encoder	5
Genbuf	16	Mult (Toy Examples)	8
HWMCC	16	Mv/Mvs (Toy Examples)	8
HyperLTL	15	Stay (Toy Examples)	8
Load Balancer (LTL2AIG)	16		
LTL2DBA/LTL2DPA (LTL2AIG)	16		
Moving Obstacle	16		
Matrix Multiplication	16	**Total:**	**250**

4.2   General Rules

Like in the previous year, SYNTCOMP is divided into two main tracks: *realizability checking* and *synthesis*, as well as into two execution modes: *sequential* and *parallel*. We explain the rules of the competition, including evaluation of tools in the different tracks.

Input and Output Format.   Participants have to accept input in the SYNTCOMP format, without further user intervention except for a fixed sequence of command-line parameters. In the synthesis track, solutions for realizable specifications also have to be provided in the SYNTCOMP format. Syntactic conformance to the format is automatically checked by a script provided by the organizers. For a description of input and output format, we refer to the report of the first competition [30].

Timeout.   In the sequential execution mode, the timeout for each problem is 3600s of CPU time. In the parallel mode, the timeout is 3600s of wall time.

Basic Ranking Scheme.   In both tracks, a correct answer is rewarded with one point for the solver, and a wrong answer is punished by subtracting 4 points. Since most of the benchmarks were available to the participants before the competition and we allowed re-submission in case of implementation bugs that were detected in the testing phase of the competition, such a punishment was not necessary. The tracks differ in how a solution is determined to be correct, as explained below.

Realizability Track.   In this track, tools read the problem description and have to return `unrealizable` in case of an unrealizable specification, and `realizable` in case of a realizable specification. Correctness is determined based on the `STATUS` information in the classification paragraph of the benchmark, unless `STATUS` is `unknown`. In the latter case, correctness is determined by a majority vote of all solvers that provide a solution to the benchmark, and the execution platform for the experiments (see Section 4.3) generates a notification that a previously unsolved problem has been solved.

Synthesis Track.   In this track, tools read the problem description and have to return `unrealizable` in case of an unrealizable specification, and a solution that satisfies the specification and confirms to the SYNTCOMP format in case of a realizable specification. In addition to the information in the `STATUS` tag of the file, correctness of a solution is checked by running a model checker on the output file, with a separate timeout of 3600s. Only solutions that can be verified by the model checker are accepted. In case of problems that have status `unknown` and are solved for the first time, in this track there is *no* majority vote on the correctness of the solution: if at least one solver produces a correct solution, it is assumed to be realizable. Similarly, if a problem is tagged with `unrealizable` and a solver produces a solution that is accepted by the model checker, we assume that the tag was wrong and the solution is correct.

Quality Ranking.   In addition to the basic ranking scheme, we define a *quality ranking* for the synthesis track, in which solutions of realizable problems are awarded a different number of points, depending on the size of the solution $size_{new}$ and a reference size $size_{ref}$. The number of points for a solution is

$$2 - \log_{10} \frac{size_{new}}{size_{ref}}.$$

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 $size_{ref}$, e.g., a solution that is smaller by a factor of 10 gets 3 points.

   For benchmark problems that have a `REF_SIZE` tag (that is not equal to 0), the value given there is taken as $size_{ref}$. For benchmark problems that do not have such a tag because they are new or have not been solved in the last competition, we use the smallest size of any of the solutions of this year as the reference size.

4.3   Execution

SYNTCOMP 2015 used a set of identical machines with a single quad-core intel Xeon processor (E3-1271 v3, 3.6GHz) and 32 GB RAM (PC1600, ECC), running a GNU/Linux system. Each node has a local 480 GB SSD that can be used as temporary storage.

As in 2014, the competition was organized on the EDACC platform [5], with a very similar setup. To ensure a high comparability and reproducability of our results, a complete machine was reserved for each job, i.e., one synthesis tool (configuration) running one benchmark. Olivier Roussel's `runsolver` [42] was used to run each job and to measure CPU time and wall time, as well as enforcing timeouts. As all nodes are identical and no other tasks were run in parallel, no other limits than a timeout of 3600 seconds (CPU time in sequential mode, wall time in parallel mode) per benchmark was set. Like last year, we used wrapper scripts to execute solvers that did not conform completely with the output format specified by the competition, e.g., to filter extra information that was displayed in addition to the specified output.

The model checker used for checking correctness of solutions is IIMC [13] in version 2.0.

5   Participants

Four participants were entered into SYNTCOMP 2015. All of them follow the traditional game-based approach to the synthesis of reactive systems from safety specifications, as described in Section 2. In this section, we briefly describe the methods implemented in each tool. Since all entrants have already participated in the competition in 2014, we focus on the changes when compared to last year's versions. For the BDD-based tools, we give an overview of the implemented methods in Table 3. For detailed explanations of the different optimizations, we refer to last year's report [30].

Table 3: Optimizations implemented in BDD-based Tools.

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

5.1   AbsSynthe 2.0: compositional algorithms for synthesis

AbsSynthe was submitted by R. Brenguier, G. A. Pérez, J.-F. Raskin, and O. Sankur from Université Libre de Bruxelles. AbsSynthe implements different BDD-based synthesis approaches, and competed in both the realizability and the synthesis track.

Overview

The new version of AbsSynthe implements different BDD-based synthesis algorithms, with and without decomposition into independent sub-games, described in more detail in [18]. All algorithms use the BDD

package CUDD, with automatic BDD reordering using the sifting heuristic. In sequential mode, three different algorithms were entered into the competition: configuration seq1 uses a standard BDD-based fixpoint computation, with partitioned transition relation and several other optimizations (see Table 3), but without compositionality. The other two configurations seq2 and seq3 use different forms of compositional reasoning, explained below. An algorithm that uses abstraction refinement is also implemented in AbsSynthe since its first version. However, this abstraction-based algorithm did not compete this year. In addition, two parallel configurations par1 and par2 entered the competition, also explained below.

Decomposing the Specification

Let us demonstrate how we decompose the error function f_{err} of a given symbolic game into a disjunction, i.e., $f_{\mathsf{err}} = \bigvee_i e_i$. Notice that if a strategy (*i.e.* a controller) ensures that the error signal is never true then it also ensures that e_i is never true. The rationale behind this approach is that the functions e_i do not depend on all latches in general, so solving the game for e_i is often efficient.

Consider an AIG representing the formula

$$x_1 \wedge \neg \left(x_2 \wedge (\neg x_3 \wedge x_4) \right),$$

where x_1, x_2, x_3, x_4 are all input variables. We can rewrite the formula as follows $\varphi_1 \equiv x_1 \wedge \neg \varphi_2$ where φ_2 is $\varphi_2 \equiv x_2 \wedge \neg x_3 \wedge x_4$. If we distribute the disjunction from $\neg \varphi_2$ we get that $\varphi_1 \equiv (x_1 \wedge \neg x_2) \vee (x_1 \wedge x_3) \vee (x_1 \wedge \neg x_4)$. Thus, one possible decomposition of φ_1 would be to take $e_1 = x \wedge \neg x_2$, $e_2 = x_1 \wedge x_3$, and $e_3 = x_1 \wedge \neg x_4$.

These general steps can be generalized into an algorithm which outputs a decomposition of the error function whenever one exists. Intuitively, the algorithm consists in exploring all non-inverted edges of the AIG graph from the vertex which defines the error function. If there are no inverted edges which stopped the exploration, or if all of them lead to leaves, the error function is in fact a conjunction of Boolean variables and can clearly not be decomposed. Otherwise, there is at least one inverted edge leading to a node representing an AND gate. In this case, we can push the negation one level down and obtain a disjunction which can be distributed to obtain our decomposition.

Given the set of e_i (*i*) we first simplify the transition relation using the classical *generalized co-factor* BDD operation and keep it precise only in for latch transition functions which affect e_i. (*ii*) Next, we solve each sub-game and obtain the set of states from which a winning strategy exists for the controller. (*iii*) When combining solved sub-games, we further modify the transition relation by making every transition not allowed by winning strategies of the controller in sub-game go to an error state.

Compositional Algorithms

We provide three different algorithms that first solve the sub-games corresponding to the sub-circuits obtained by our decomposition procedure, and then aggregate, following three different heuristics, the results obtained on the sub-games. Namely, once we have the solutions of all the sub-games we aggregate them by using one of the three heuristics described below.

Global aggregation. We start by computing the intersection of the winning valuations of all sub-games: $\Lambda = \bigwedge_{1 \leq i \leq n} w_i(L, X_u, X_c)$, where each $w_i(L, X_u, X_c)$ is a winning valuation of the latches L, uncontrollable inputs X_u, and controllable inputs X_c, for subgame i. In fact, any valuation that is not in Λ is losing in one of the sub-games; thus in the global game. Conversely, a strategy that stays in Λ is winning for each sub-game. Therefore, we solve the global game with the new safety objective of avoiding $\neg \Lambda$.

Before solving the global game, the algorithm attempts to reduce the size of the transition relations using *restrict*. This algorithm entered SYNTCOMP 2015 as configuration seq2.

Incremental aggregation.   In this algorithm, we aggregate the results of the sub-games *incrementally*: given the list of winning valuations w_i for the sub-games, at each iteration, we choose and remove two sub-games i and j, solve their conjunction (as with global aggregation, with error function $\neg(w_i \wedge w_j)$), and add the newly obtained winning valuations back in the list. The choice of the pair i, j is done heuristically. This algorithm entered SYNTCOMP 2015 as configuration seq3.

Back-and-forth.   For this last algorithm, we interleave the analysis of the global game (with objective Λ) and the analysis of the sub-games. Let $u(L)$ be the set of "bad" states (initially containing all states where err is true). At each iteration, we extend the losing states $u(L)$ by one step, by applying once the upre operator in the global space. We then consider each sub-game, and check whether the new set $u'(L)$ of losing states (projected on the sub-game), changes the local winning states. For those games in which it does grow, we recompute the local fixed point. We update the strategies λ_i of the sub-games when necessary, and restart until stabilization. Because analyzing the sub-games is often more efficient than analyzing the global game, this algorithm improves over the global aggregation algorithm in some cases (see the experiments' section). A similar idea was used in [27] for the problem of synthesis from LTL specifications. This algorithm did not enter SYNTCOMP 2015 as a separate configuration in sequential mode, but was used as one thread in one of the parallel modes, see below.

Parallel Algorithms.   Two parallel versions of AbsSynthe entered the competition. Configuration par1 launches four threads in parallel, which execute the algorithms of sequential configurations seq1, seq2 and seq3, plus another thread with back-and-forth compositional reasoning. Configuration par2 also launches four threads in parallel, but in this case with four instances of strategy seq1 that only differ in the reordering strategy for BDDs.

Strategy Extraction

Strategy extraction in AbsSynthe uses the co-factor-based approach described in [11], with some additional optimizations as described in [30].

Experiments

Amongst the 674 benchmarks considered in [18] (which include those used in SYNTCOMP 2014), 351 are decomposable by our static analysis into at least 2 smaller sub-games. More specifically, the average number of sub-games our decomposition algorithm outputs is 29; the median is 21.

In general, the performances of the three compositional algorithms can differ, but they are complementary. Even if AIG synthesis problems are monolithic, the experiments show that our compositional approach was able to solve problems that can not be handled by the monolithic backward algorithm; our compositional algorithms are sometimes much more efficient. There are also examples that can be decomposed but which are not solved more efficiently by the compositional algorithms. So, it is often a good idea to apply all the algorithms in parallel.
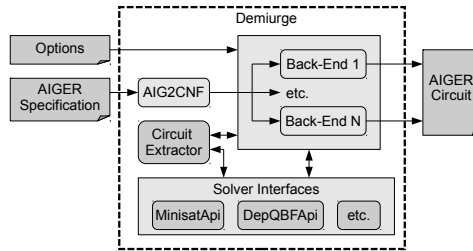
Figure 3: The architecture of Demiurge.

Implementation, Availability

AbsSynthe is implemented in C++, and depends only on a simple AIG library (fetched from [6]) and the CUDD binary decision diagram package [46]. The source code is available at `https://github.com/gaperez64/AbsSynthe/tree/native-dev-par`. More details on AbsSynthe can be found in [17,18].

5.2 Demiurge 1.2.0: A SAT-Based Synthesis Tool

Demiurge was submitted by R. Könighofer from Graz University of Technology and M. Seidl from Johannes-Kepler-University Linz. Demiurge implements several symbolic synthesis algorithms based on SAT and QBF solvers. Demiurge competed in both the realizability and the synthesis track.

Overview

The architecture of Demiurge is outlined in Fig. 3. The input is a safety specification in AIGER format. The AIG2CNF module parses it into CNF formulas representing the transition relation and the set of safe states. Next, the back-end selected by the user is executed. The back-ends mostly differ in their method for computing the winning region, and can be parameterized with a method for computing a circuit from the winning region. Both the computation of the winning region and the extraction of circuits rely on external solvers like SAT- and QBF solvers. The resulting circuits are optimized with ABC [16] and dumped in AIGER format again.

Back-Ends

Learning-Based Back-End. The learning-based back-end computes a CNF representation of the winning region W in an iterative manner. It starts with the set of all safe states. In each iteration, it computes a state within the current version F of the winning region from which the environment can enforce to leave F. Obviously, such a state cannot be part of the final winning region W. Hence, the algorithm refines F by removing this state. The state is represented as a cube over the state variables, so removing it from F amounts to adding a clause. By dropping literals from the cube as long as it only contains states that must be excluded from the winning region, the algorithm generalizes the state into a larger region before removing it from the winning region. The detailed algorithm can be found in [10].

For SYNTCOMP 2015, we use the following configuration, called D1synt if it is called with circuit extraction, and D1real otherwise. Instead of a QBF solver, we use two competing SAT solvers to compute and generalize states to be removed from the winning region (algorithm LEARNSAT from [10] with optimization RG enabled, but optimization RC disabled). As a difference to the SYNTCOMP 2014

submission (version 1.1.0), we also apply partial universal quantifier expansion to reduce the number of iterations. Minisat version 2.2.0 is the underlying SAT solver.

Template-Based Back-End. In order to obtain a winning region, this back-end constructs a parameterized CNF formula over the state variables: different concrete values for the (Boolean) parameters induce a different concrete CNF formula over the state variables. This way, the search for a formula over the state variables (the winning region) is reduced to a search for Boolean constants (the template parameter values) [10]. While Demiurge 1.1.0 could only use a QBF solver for finding a template instantiation, version 1.2.0 can also use a SAT solver in a Counterexample-Guided Inductive Synthesis (CEGIS) approach. For SYNTCOMP 2015, this back-end is not run separately, but only as one thread in our parallelization.

Incremental Induction Back-End. The *incremental induction back-end* is a re-implementation of [38], generalizing IC3-based reachability checking [14] to the synthesis case. For SYNTCOMP 2015, this back-end is not run separately, but only as one thread in our parallelization.

Parallel Back-End. The parallel back-end is a playground for combining different methods that refine a CNF representation of the winning region iteratively with additional clauses. Several threads compute and add additional clauses in parallel.

For SYNTCOMP 2015, we use configurations with 3 threads, called P3synt or P3real for the synthesis and realizability tracks, respectively: One thread executes the learning-based back-end, one the template-based back-end (alternating between QBF- and SAT solving in 20 second turns), and one our incremental induction back-end. Minisat 2.2.0 is used as SAT solver. DepQBF 3.04 with our extension of the QBF preprocessor Bloqqer [45] is used for QBF solving in the template-based thread. Note that our parallelization is not just a portfolio approach. The different threads share clauses of the winning region as soon as they are discovered such that other threads can immediately benefit from this information.

Circuit Extraction

Demiurge provides several methods for computing circuits from the winning region [8]. One uses QBFCert to compute Skolem functions for the output signals in a QBF that asserts completeness of the strategy relation. The second one uses computational learning to compute circuits for one output after the other. A third method is based on interpolation.

For SYNTCOMP 2015, we use the learning approach (method SL from [8]) with Lingeling ayv as SAT solver. In our parallelization, we use 3 threads. The first two execute the learning approach in two variants (SL and SLN from [8]) . The third thread executes the learning approach using incremental QBF solving (method QL from [8]) with DepQBF 3.04.

Implementation, Availability

Demiurge is implemented in C++, and depends on a number of underlying reasoning engines. Currently, Demiurge contains uniform interfaces to the APIs of Minisat, Lingeling, PicoSat, and DepQBF (with and without the QBF preprocessor Bloqqer [45]). The interface to DepQBF also supports incremental QBF solving [35]. Interfaces to SAT and QBF solvers supporting the (Q)DIMACS format are available

as well. Furthermore, Demiurge interfaces ABC [16] for circuit minimization. The source code is available at `http://www.iaik.tugraz.at/content/research/design_verification/demiurge/` under the GNU Lesser General Public License version 3. The downloadable archive also contains extensive experimental results on the SYNTCOMP 2014 benchmarks and scripts to reproduce them.

5.3   Realizer

Realizer was submitted by L. Tentrup from Saarland University, Saarbrücken. Realizer implements BDD-based realizability checking, and competed in the realizability track. It does not support extraction of strategies. We only give a very brief description of Realizer, since there are only minor changes to the version that competed in SYNTCOMP 2014 [30].

Synthesis algorithms

Realizer implements the standard BDD-based fixpoint algorithm for safety games. It is based on BDD package CUDD, and uses automatic reordering of BDDs with the *lazy sift* reordering scheme. The fix-point algorithm is implemented in two variants, differing only in the way they handle the transition relation of the circuit: one variant uses a monolithic transition relation, while the other uses a partitioned transition relation. Only the variant with partitioned transition relation competed in sequential execution mode, as the other one is not competitive in general. A number of additional optimizations are implemented in Realizer, compare Table 3.

   The main difference (besides bug fixes) to the version that competed in 2014 is the *parallel mode*, which uses both variants of the algorithm running (independently) in parallel.

Implementation

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.

5.4   Simple BDD Solver 2

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 realizability subtrack. It does not support extraction of strategies.

Overview

Simple BDD Solver is a substantial simplification of the solver that was developed for the Termite project (`http://termite2.org`), adapted to safety games given in the AIGER format. It uses the BDD package CUDD, with dynamic variable reordering using the sifting algorithm [43], and a number of additional optimizations (again, cp. Table 3 and [30]). The basic algorithm entered the competition as configuration 1, and is almost identical to the version that competed last year.

   Furthermore, the tool implements a variant of the fixpoint algorithm with an abstraction-refinement loop inspired by de Alfaro and Roy [3]. This variant did not compete last year, but entered the competition this year. This algorithm entered the competition as configuration 2.

Abstraction Refinement for Synthesis

The idea of the abstraction-refinement algorithm is that, given an abstraction, we classify states into one of three categories: winning, losing, and unknown. If we discover that the entire initial set is winning, we know that the original game is winning and we can terminate. Dually, if we discover any initial state that is losing, we know that the entire initial set can never be winning, hence the game is losing and we can terminate.

The algorithm iteratively refines the abstraction to reduce the number of states in the unknown classification until either:

- all of the initial states are classified as winning (and the other states need not be classified, since this means that we can win from all initial states), or

- one of the initial states is classified as losing (again, no other states need to be classified)

The solver creates an abstraction by dropping a subset of state variables from the transition relation and instead allows them to be non-deterministically updated on each round of the game. The initial abstraction consists of only the variables that are mentioned in the safety specification, i.e., those that the error output err directly depends on.

Abstract Game Solving. The abstract game is based on two different interpretations of the transition relation, that compute the *controllable predecessors* of a set of states with different assumptions: Cpre-must ($Cpre_1^M$) resolves the non-determinism of the abstraction in favor of the environment player, i.e., we assume that the system player can only force the game from an abstract state s into an abstract state s' if this is possible for *any* valuation of the state variables that have been abstracted away. On the other hand, Cpre-may ($Cpre_1^m$) resolves the abstraction in favor of the system player, i.e., we assume that the system player can force the game from s into s' if there exists a valuation of the abstracted variables such that this is possible.

Based on these functions, one can compute may-winning and must-winning regions, denoted W^m and W^M respectively, by iterating the functions above to a fixpoint.

Abstraction Refinement. Following de Alfaro and Roy [3], we refine the abstraction at the may-must boundary, i.e. the boundary between states classified as winning and unknown. We split an abstract state into two such that from one of the new states x is is possible for the environment player to force execution into the losing region. We find a candidate state to split by using an efficient symbolic calculation that determines the sub-states that are not must-winning but have a transition to the must-winning set W^M. These are the sub-states on the may-must boundary that are winning and are thus are part of a candidate state for splitting.

In order to extract a single sub-state on the may-must boundary that is winning, we extract a large prime implicant from this characteristic formula. This is an efficient operation with BDDs. We also extract the variables that occur in this prime that and are currently abstracted away.

Notice that if we include these variables in our abstraction, the abstraction is now precise enough to exactly represent the states in the implicant. These states are winning. This means that if we iterate the controllable predecessor one more time, we will at least discover that this implicant is winning, in addition to W^M. Thus, our refinement has grown the must winning set. In practice, refining the abstraction in this way usually discovers many more winning states than just the implicant.

Implementation, Availability

Simple BDD solver is written in the Haskell functional programming language. It uses the CUDD package for BDD manipulation and the Attoparsec [39] 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`.

6   Experimental Results

We present the results of SYNTCOMP 2015, separated into realizability and synthesis track. Detailed results of the competition are also directly accessible via the web-frontend of the EDACC platform at `syntcomp.cs.uni-saarland.de`.

6.1   Realizability Track

In the realizability track, tools competed on 250 benchmark instances, selected from the different benchmark categories as explained in Section 4.1. All 4 tools that entered SYNTCOMP 2015 competed in the realizability track in at least one configuration. Overall, 10 different configurations entered this track, with 7 using sequential execution mode and 3 using parallel mode. We first restrict the evaluation of results to purely sequential tools, then extend it to include also the parallel versions, and finally give a brief analysis of the results.

Sequential Mode.   In sequential mode, all participants compete with at least one configuration: AbsSynthe with three configurations (seq1, seq2 and seq3), Demiurge with one configuration (D1real), Realizer with one configuration (sequential), and Simple BDD Solver with two configurations (1 and 2).

The number of solved instances per configuration, as well as the number of uniquely solved instances, are given in Table 4. No tool could solve more than 200 out of the 250 instances, and 30 instances could not be solved by any tool within the timeout.

Table 4: Results: Realizability (sequential mode only)

Tool (configuration)	Solved	Unique
Simple BDD Solver (2)	195	9
AbsSynthe (seq2)	187	2
Simple BDD Solver (1)	185	0
AbsSynthe (seq3)	179	0
Realizer (sequential)	179	0
AbsSynthe (seq1)	173	1
Demiurge (D1real)	139	5

The following benchmarks were solved uniquely by one tool configuration:

- AbsSynthe (seq1): `moving_obstacle_128x128_59glitches`

- AbsSynthe (seq2): `mult_bool_matrix_6_6_6, mult12`

- Demiurge (D1real): `beemldelec4b1_c0to511`, `gb_s2_r2_comp4_REAL`, `load_full_4_comp1_REAL`, `mult_bool_matrix_dyn_6_6`, `very_good_bakery2.sym`
- Simple BDD Solver (2): `amba9match5`, `cycle_sched_12_6_3`, `cycle_sched_8_7_2`, `driver_a7n`, `driver_b7y`, `driver_c8n`, `factory_assembly_5x5_2_10errors`, `factory_assembly_5x5_2_11errors`, `good_bakery.false`

For comparison, we also ran a number of additional tools on the benchmark set: last year's versions of AbsSynthe, Simple BDD Solver, the learning-based sequential version of Demiurge, as well as Aisy [32], our BDD-based, unoptimized reference implementation.[1] The results for these tools can be found in Table 5. Furthermore, Figure 4 gives a cactus plot for runtimes of sequential algorithms in the realizability track, including the reference implementation Aisy.

Table 5: Results: Realizability (sequential, reference solvers)

Tool (configuration)	Solved
Simple BDD Solver (2014)	182
AbsSynthe (2014)	169
Demiurge (learn,2014)	102
Aisy	98
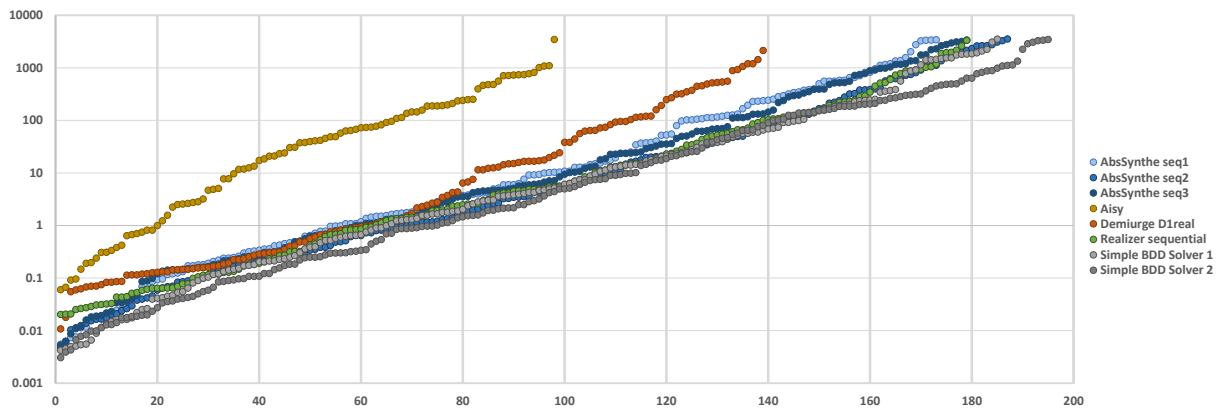
Figure 4: Sequential Realizability Track, Runtime Cactus Plot

Parallel Mode. Three of the tools that entered the competition had at least one parallel configuration for the realizability track: two configurations of AbsSynthe (par1 and par2), and one configuration each of Demiurge (P3real) and Realizer (parallel). The difference to sequential mode is that the tools can use all four cores of the CPU, and the timeout is now measured in Wall Time instead of CPU Time. In particular, the parallel configurations had to solve the same set of benchmark instances as in the sequential mode. The results are given in Table 6. Again, no tool could solve more than 200 instances, but a number of additional instances could be solved: only 11 could not be solved by any tool in either sequential or parallel mode.

---

[1]We don't consider last year's version of Realizer, since the improvements in the new version are limited to bug-fixes and the new parallel mode.

Table 6: Results: Realizability (parallel mode only)

Tool (configuration)	Solved	Unique
AbsSynthe (par1)	193	0
Realizer (parallel)	185	3
Demiurge (P3real)	183	17
AbsSynthe (par2)	173	0

For an analysis of benchmark instances by category, Figures 5, 6 and 7 give an overview of the number of solved instances per configuration and category.
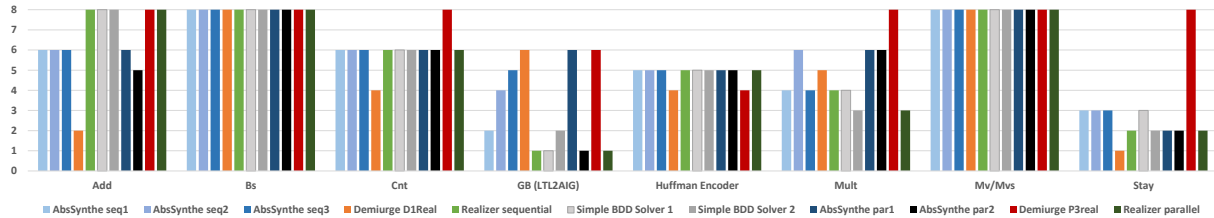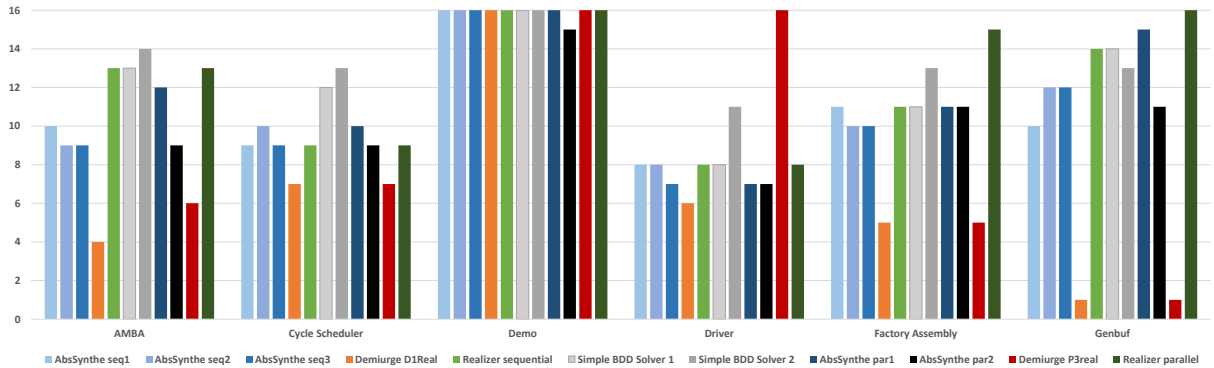

Figure 5: Realizability Track, Solved Instances by Category

Figure 6: Realizability Track, Solved Instances by Category

Note also that in Table 6 we only count a benchmark instance as uniquely solved if it is not solved by any other configuration, including the sequential configurations. Considering both sequential and parallel configurations, the following benchmarks were solved uniquely by one tool configuration:

- Demiurge (P3real): `6s216rb0_c0to31.aag`, `cnt30n.aag`, `cnt30y.aag`, `driver_a10n.aag`, `driver_a8n.aag`, `driver_b10y.aag`, `driver_b8y.aag`, `driver_c10n.aag`, `mult14.aag`, `mult16.aag`, `oski2ub1i_c0to7.aag`, `oski3ub1i_c0to255.aag`, `stay18y.aag`, `stay20n.aag`, `stay20y.aag`, `stay22n.aag`, `stay22y.aag`,

- Realizer (parallel): `factory_assembly_7x5_2_10errors.aag`, `factory_assembly_7x5_2_11errors.aag`, `genbuf64c2unrealy.aag`,

- Simple BDD Solver (2): `cycle_sched_12_6_3.aag`, `cycle_sched_8_7_2.aag`.
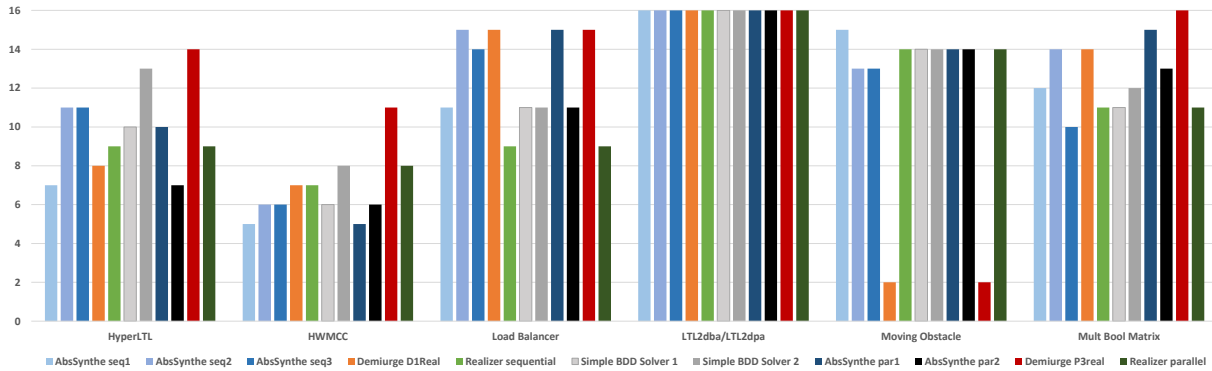
Figure 7: Realizability Track, Solved Instances by Category

Considering again last year's versions, only Demiurge ran in a proper parallel mode in 2014.[2] Out of the 250 benchmark instances, Demiurge (parallel, 2014) could solve 99 instances. This is slightly less than the sequential configuration from 2014, as is to be expected based on last year's results [30].

Finally, to further investigate the improvement of AbsSynthe, Demiurge, and Simple BDD Solver compared to last year's versions, we give additional cactus plots that compare the best-performing configurations from each tool from 2014 and 2015 in Figures 8, 9, and 10.
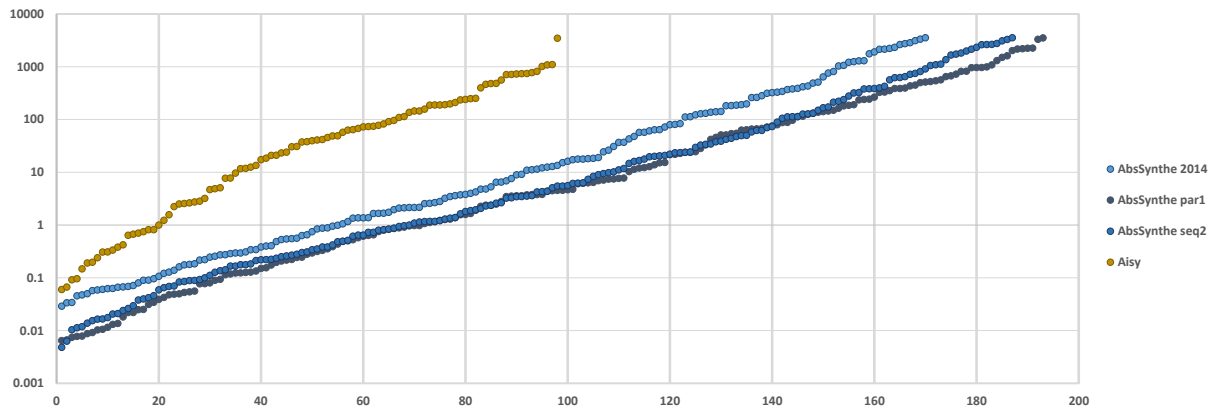
Figure 8: Realizability Track, Improvement of AbsSynthe over Last Year

Analysis. We note that on this year's benchmark set, the sequential configuration AbsSynthe (seq2) solve slightly more problems than last year's winner, represented by configuration Simple BDD Solver (1). However, the new abstraction-based configuration Simple BDD Solver (2) can solve 10 additional problems, which is slightly more than the best parallel version of AbsSynthe, configuration (par1).

Comparing the participants to the unoptimized reference implementation Aisy, we see in Figure 4 that the BDD-based participants can be considered to be two orders of magnitude faster than Aisy, and in Figure 9 that Demiurge (P3real) is faster by three orders of magnitude on a lot of problems.

Considering the different configurations of AbsSynthe, we note that the best-performing sequential configuration is (seq2), which uses the compositional algorithm with global aggregation. Configuration

---

[2]AbsSynthe and Realizer had parallel modes, but due to bugs and faulty call parameters did not produce interesting results.
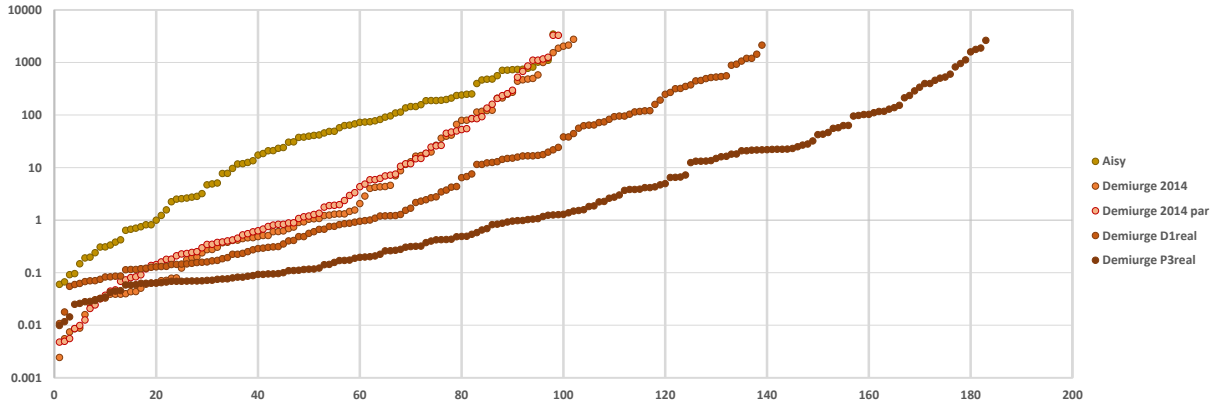
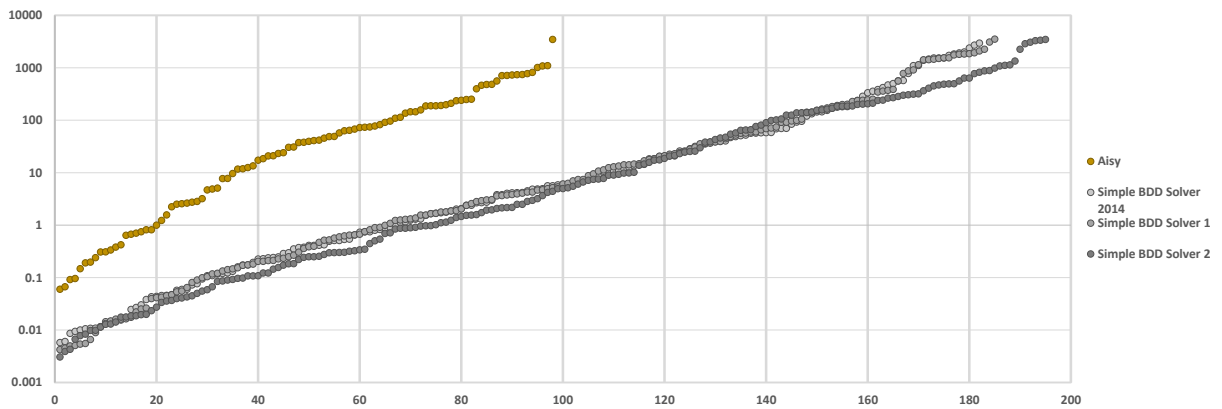Figure 9: Realizability Track, Improvement of Demiurge over Last Year

Figure 10: Realizability Track, Improvement of Simple BDD Solver over Last Year

(seq1), which is essentially the same as last year's version, solves the least number of problems. The best configuration overall is the parallel configuration (par1), which runs the different compositional algorithms as well as the non-compositional algorithm (seq1) in parallel. Running multiple threads of (seq1) in parallel, only with different BDD reordering strategies, did not pay off: configuration (par2) even solves less problems than (seq1).

Finally, Demiurge shows an impressive improvement over last year's version: the sequential configuration Demiurge (D1real) can solve 37 additional instances when compared to last year's sequential (learn) configuration, and the new parallel mode solves 44 more problems than the sequential configuration, or 86 more problems than last year's parallel configuration. Comparing it to the other tools on different benchmark classes (see Figures 5, 6 and 7), we note that it still cannot compete with the other tools on benchmark classes such as AMBA, Factory Assembly, Genbuf, and Moving Obstacle, but it outperforms all other tools on benchmark classes Cnt, Mult, Stay, Driver, HyperLTL, HWMCC, and Mult Bool Matrix, most of which are new benchmarks that have been added this year.

6.2 Synthesis Track

In the synthesis track, tools competed on the same benchmarks as in the realizability track, except that those instances that could not be solved by any configuration (sequential or parallel) in the realizability track have been removed from the benchmark set. Thus, the benchmark set for the synthesis track contains 239 instances. Two tools entered the synthesis track: AbsSynthe with three sequential and two parallel configurations, and Demiurge with one configuration for each mode.

In the synthesis track, we have two different ranking schemes: the basic ranking is by number of problem instances that can be solved within the timeout, and the separate *quality ranking* gives more points to small solutions of realizable specifications, as explained in Section 4.2. In both cases, a solution for a realizable specification is only considered as correct if it can be model-checked within a separate timeout of one hour (cf. Sections 4.2 and 4.3). Like in the realizability track, in the following we start by presenting the results for the sequential configurations, followed by parallel configurations, and end with an analysis of the results.

Sequential Mode. In this mode, AbsSynthe competed with three configurations (seq1, seq2, seq3), and Demiurge with one configuration (D1synt). In addition, we ran last year's versions of AbsSynthe and the learning-based configuration of Demiurge, as well as our reference implementation Aisy, on this year's competition benchmarks.

Table 7 summarizes the experimental results, including the number of solved benchmarks, the number of points in the quality ranking, the uniquely solved instances, and the number of solutions that could not be model-checked within the timeout. Note that in the table a (realizable) problem instance is only considered as solved if the tool presents a solution that can be model-checked. With this requirement, no tool could solve more than 161 or about 67% of the benchmarks, and 60 instances could not be solved by any tool. In particular, for all configurations of AbsSynthe there is a rather high number of benchmarks for which the tool can provide a solution, but this solution cannot be model-checked.

Table 7: Results: Synthesis (sequential mode only)

Tool (configuration)	Solved	Quality	Unique	MC Timeout
AbsSynthe (seq2)	161	254	4	16
AbsSynthe (seq3)	152	241	1	16
AbsSynthe (seq1)	148	234	6	18
AbsSynthe (2014)	145	231	%	16
Demiurge (D1synt)	127	214	8	4
Demiurge (2014,learn)	83	138	%	1
Aisy	75	105	%	3

Parallel Mode. In this mode, AbsSynthe competed with two configurations (par1, par2), and Demiurge with one configuration (P3synt). In addition, we ran last year's parallel configuration of Demiurge on the new competition benchmarks.

Table 8 summarizes the experimental results, again including the number of solved benchmarks, the number of points in the quality ranking, the uniquely solved instances, and the number of solutions that could not be model-checked within the timeout. No tool could solve more than 180 problem instances,

or about 75% of the benchmark set. The number of (potential) solutions that cannot be model checked within the timeout is again rather high for AbsSynthe, while Demiurge only produces a single solution that cannot be checked. Like in the parallel realizability track, we only consider instances as uniquely solved if they are not solved by any other configuration, including sequential ones.

Table 8: Results: Synthesis (parallel mode only)

Tool (configuration)	Solved	Quality	Unique	MC Timeout
Demiurge (P3Synt)	180	317	28	1
AbsSynthe (par1)	167	263	2	20
AbsSynthe (par2)	148	235	0	17
Demiurge (2014,parallel)	88	144	0	1

Analysis. Consider the fact that all configurations of AbsSynthe produce a high number of (potential) solutions that cannot be model checked. In contrast, Demiurge only produces very few such solutions. We assume that this is in part due to the fact that Demiurge in most cases produces smaller solutions than AbsSynthe. We compare solution sizes for some problem instances in Figures 11, 12 and 13.
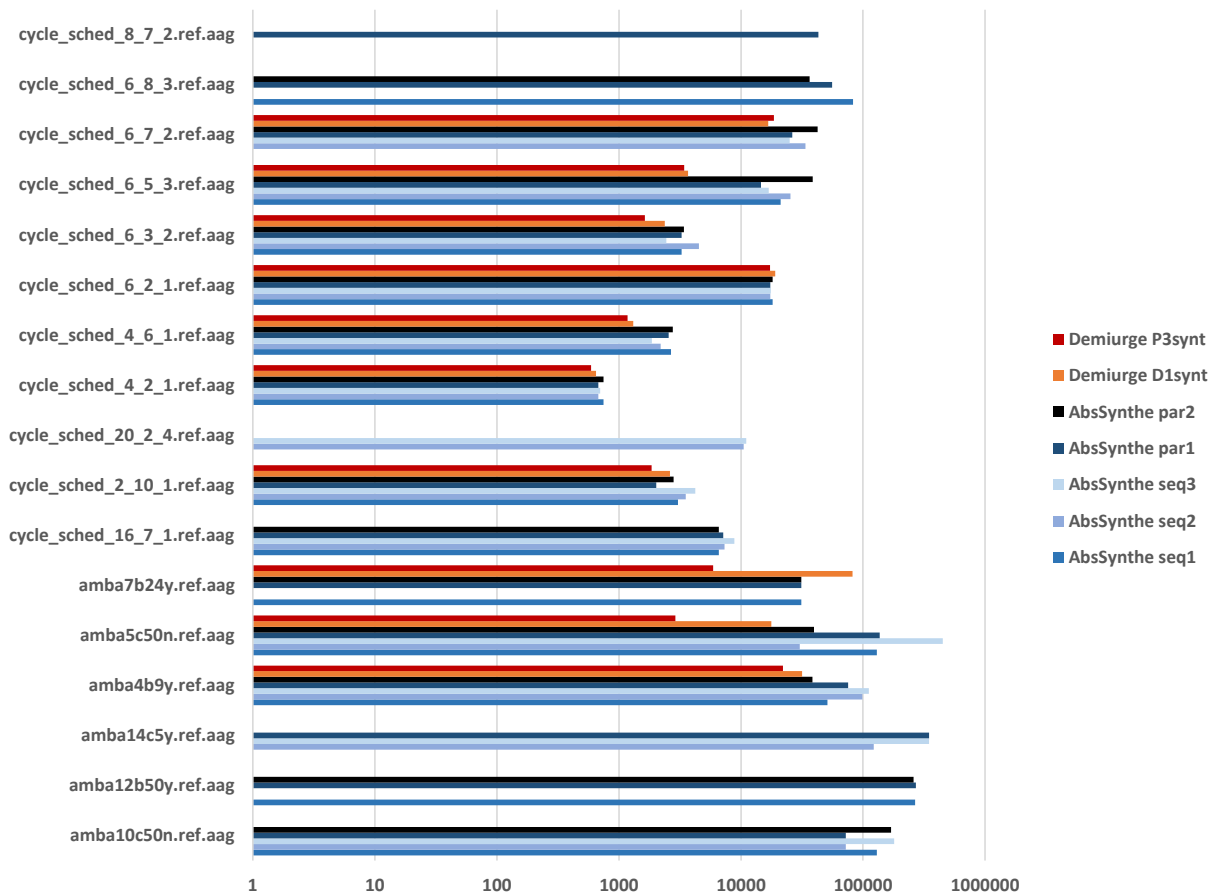
Figure 11: Comparison of Solution Size for AMBA and Cycle Scheduling Benchmarks
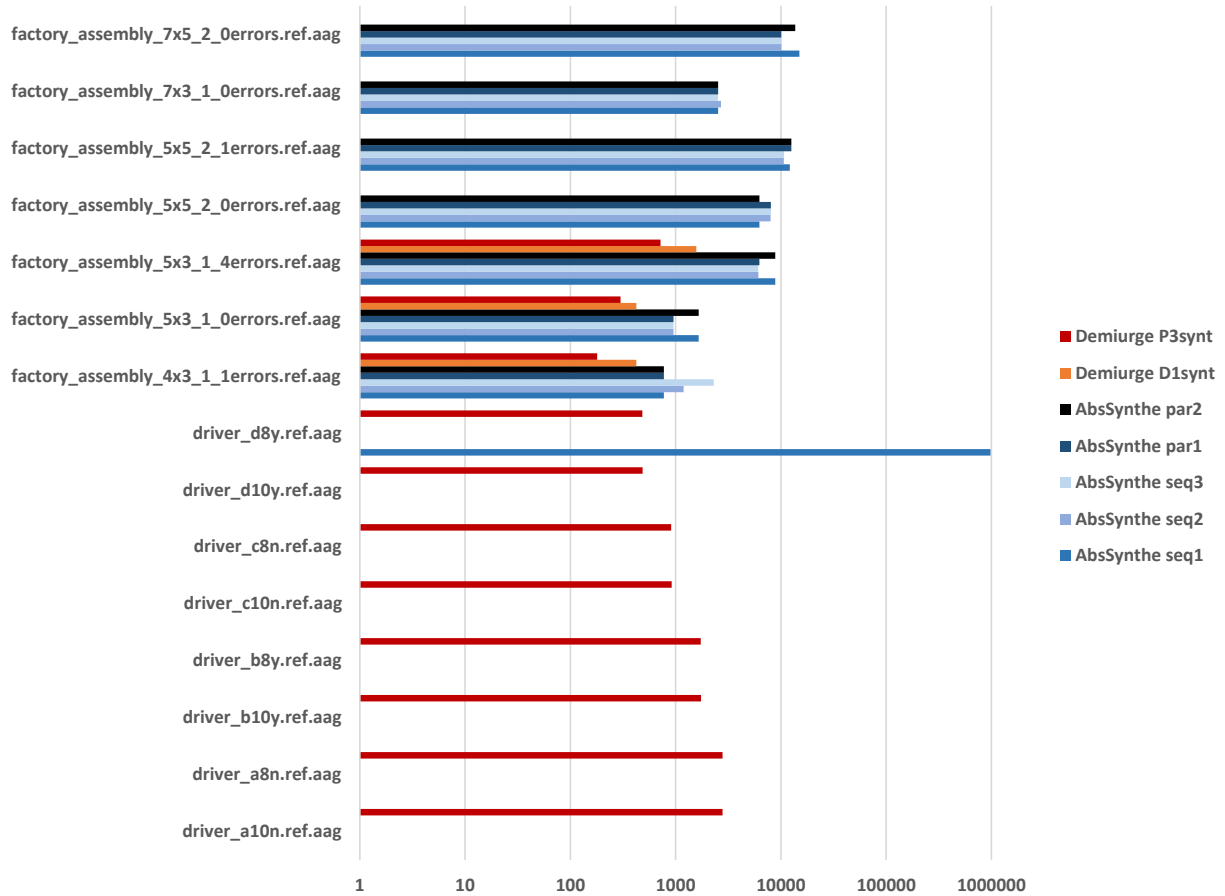
Figure 12: Comparison of Solution Size for AMBA and Cycle Scheduling Benchmarks

Apart from the issue of non-verifiable solutions, the performance of the different algorithms is exactly what is to be expected, based on the results of the realizability track. In particular, note that without the requirement of verifiable solutions, the parallel configuration AbsSynthe (par1) would have ranked higher than Demiurge (P3synt).[3]

Also note that, although our quality metric does not change the ranking of tools, it does increase the relative difference in some cases. E.g., the difference between Demiurge (P3synt) and AbsSynthe (par1) is only 8% in number of solved instances, but 20% in the quality score. An analysis of the size of the potential solutions that could not be model checked shows that none of them is smaller than the reference size, which implies that AbsSynthe (par1) would be ranked below Demiurge (P3synt) in the quality ranking, even if all of its solutions could be model checked.

Considering uniquely solved instances, we note that the differences are rather small when we only compare the sequential configurations — each configuration produces some unique solutions, and Abs-Synthe (seq1) produces almost as many as Demiurge (D1synt). This picture changes when we look at the parallel configurations: Demiurge (P3synt) has a high number of uniquely solved instances, i.e., instances that can neither be solved by any sequential configuration, nor by the parallel configurations of

---

[3]We tried to resolve the problem of model checking timeouts by running other model checkers on a number of the solutions that cannot be verified. In particular, we tried the other two highest-ranking tools for single safety properties in HWMCC 2014, ABC [16] and V3 [2, 48] However, neither of the tools was successful on the solutions that we tried to verify.
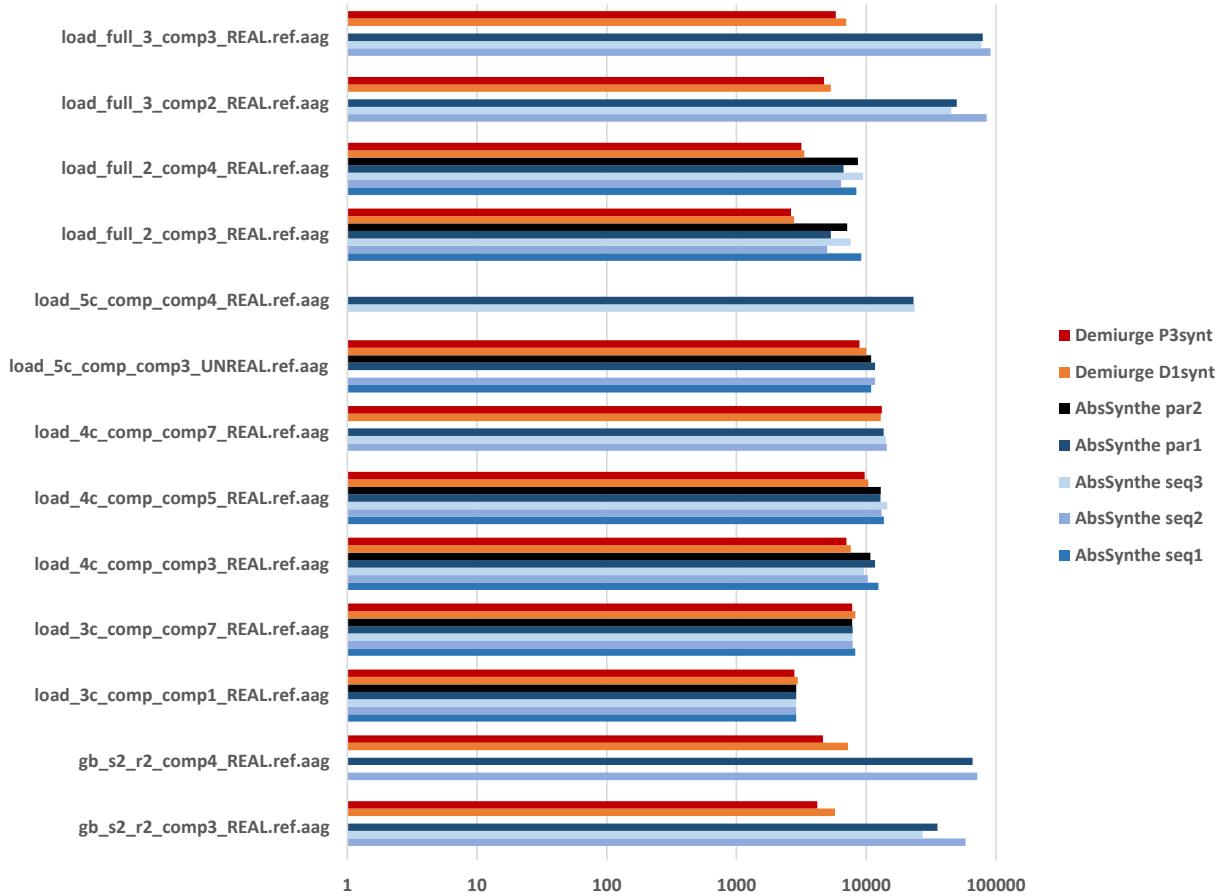
Figure 13: Comparison of Solution Size for AMBA and Cycle Scheduling Benchmarks

AbsSynthe. In contrast, almost all instances that are solved by AbsSynthe (par1) and AbsSynthe (par2) are also solved by the sequential configurations, or by Demiurge (P3synt).

7   Conclusions

SYNTCOMP 2015 was another big step towards establishing the competition in the synthesis community and extending the benchmark format and library. We have collected thousands of additional benchmark instances, and refined our ranking and evaluation for a fairer comparison of participants. In addition, our experimental results show that some of the tools have made impressive improvements when compared to last year's versions.

For 2016, we will consider an extension of the problem from pure safety specifications to a specification format that includes liveness, possibly in the form of LTL formulas.

References

[1] *Termite Driver Synthesis Tool*. http://ssrg.nicta.com.au/projects/TS/drivers/synthesis/.

[2] *V3 - An Extensible Framework for Hardware Verification*. http://dvlab.ee.ntu.edu.tw/~publication/V3/.

[3] Luca de Alfaro & Pritam Roy (2007): *Solving Games Via Three-Valued Abstraction Refinement*. In: *CONCUR*, *LNCS* 4703, Springer, pp. 74–89, doi:10.1007/978-3-540-74407-8_6.

[4] Rajeev Alur, P. Madhusudan & Wonhong Nam (2005): *Symbolic computational techniques for solving games*. *STTT* 7(2), pp. 118–128, doi:10.1007/s10009-004-0179-0.

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

[6] Armin Biere: *AIGER Format and Toolbox*. http://fmv.jku.at/aiger/.

[7] Armin Biere: *Hardware Model Checking Competition*. http://fmv.jku.at/hwmcc/.

[8] R. Bloem, U. Egly, P. Klampfl, R. Könighofer & F. Lonsing (2014): *SAT-based methods for circuit synthesis*. In: *FMCAD'14*, IEEE, pp. 31–34, doi:10.1109/FMCAD.2014.6987592.

[9] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli & Y. Sa'ar (2012): *Synthesis of Reactive(1) designs*. *J. Comput. Syst. Sci.* 78(3), pp. 911–938, doi:10.1016/j.jcss.2011.08.007.

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

[11] Roderick Bloem, Stefan J. Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli & Martin Weiglhofer (2007): *Specify, Compile, Run: Hardware from PSL*. *Electr. Notes Theor. Comput. Sci.* 190(4), pp. 3–16, doi:10.1016/j.entcs.2007.09.004.

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

[13] Aaron Bradley, Fabio Somenzi, Michael Dooley, Zyad Hassan & Yan Zhang: *Incremental Inductive Model Checker*. http://ecee.colorado.edu/wpmu/iimc/.

[14] Aaron R. Bradley (2011): *SAT-Based Model Checking without Unrolling*. In: *VMCAI*, *LNCS* 6538, Springer, pp. 70–87, doi:10.1007/978-3-642-18275-4_7.

[15] Robert K. Brayton, Gary D. Hachtel, Alberto L. Sangiovanni-Vincentelli, Fabio Somenzi, Adnan Aziz, Szu-Tsung Cheng, Stephen A. Edwards, Sunil P. Khatri, Yuji Kukimoto, Abelardo Pardo, Shaz Qadeer, Rajeev K.

Ranjan, Shaker Sarwary, Thomas R. Shiple, Gitanjali Swamy & Tiziano Villa (1996): *VIS: A System for Verification and Synthesis*. In: *CAV*, *LNCS* 1102, Springer, pp. 428–432, doi:10.1007/3-540-61474-5_95.

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

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

[18] Romain Brenguier, Guillermo A. Pérez, Jean-François Raskin & Ocan Sankur (2015): *Compositional Algorithms for Succinct Safety Games*. In: *SYNT*, this volume of *Electronic Proceedings in Theoretical Computer Science*, Open Publishing Association, Open Publishing Association.

[19] Randal E. Bryant (1986): *Graph-Based Algorithms for Boolean Function Manipulation*. IEEE Trans. Computers 35(8), pp. 677–691, doi:10.1109/TC.1986.1676819.

[20] J.R. Büchi & L.H. Landweber (1969): *Solving sequential conditions by finite-state strategies*. Trans. Amer. Math. Soc. 138, pp. 295–311, doi:10.2307/1994916.

[21] Yushan Chen, Xu Chu Ding, Alin Stefanescu & Calin Belta (2012): *Formal Approach to the Deployment of Distributed Robotic Teams*. IEEE Transactions on Robotics 28(1), pp. 158–171, doi:10.1109/TRO.2011.2163434.

[22] Sandeep Chinchali, Scott C. Livingston, Ufuk Topcu, Joel W. Burdick & Richard M. Murray (2012): *Towards formal synthesis of reactive controllers for dexterous robotic manipulation*. In: *ICRA*, IEEE, pp. 5183–5189, doi:10.1109/ICRA.2012.6225257.

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

[24] Rüdiger Ehlers (2011): *Unbeast: Symbolic Bounded Synthesis*. In: *TACAS*, *LNCS* 6605, Springer, pp. 272–275, doi:10.1007/978-3-642-19835-9_25.

[25] E. Allen Emerson & Edmund M. Clarke (1982): *Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons*. Sci. Comput. Program. 2(3), pp. 241–266, doi:10.1016/0167-6423(83)90017-5.

[26] Emmanuel Filiot: *Acacia+*. http://lit2.ulb.ac.be/acaciaplus/.

[27] Emmanuel Filiot, Naiyong Jin & Jean-François Raskin (2010): *Compositional Algorithms for LTL Synthesis*. In: *ATVA*, *LNCS* 6252, Springer, pp. 112–127, doi:10.1007/978-3-642-15643-4_10.

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

[29] David A. Huffman (1952): *A Method for the Construction of Minimum-Redundancy Codes*. Proceedings of the IRE 40(9), pp. 1098–1101, doi:10.1109/JRPROC.1952.273898.

[30] Swen Jacobs, Roderick Bloem, Romain Brenguier, Rüdiger Ehlers, Timotheus Hell, Robert Könighofer, Guillermo A. Pérez, Jean-François Raskin, Leonid Ryzhyk, Ocan Sankur, Martina Seidl, Leander Tentrup & Adam Walker (2015): *The First Reactive Synthesis Competition (SYNTCOMP 2014)*. CoRR abs/1506.08726. Available at http://arxiv.org/abs/1506.08726.

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

[32] Ayrat Khalimov: *Aisy: Simple synthesis tool from AIGER circuits format*. https://bitbucket.org/art_haali/aisy-classroom.

[33] Ayrat Khalimov (2015): *Specification Format for Reactive Synthesis Problems*. In: *SYNT*, this volume of *Electronic Proceedings in Theoretical Computer Science*, Open Publishing Association, Open Publishing Association.

[34] Hadas Kress-Gazit, Georgios E. Fainekos & George J. Pappas (2009): *Temporal-Logic-Based Reactive Mission and Motion Planning*. IEEE Transactions on Robotics 25(6), pp. 1370–1381, doi:10.1109/TRO.2009.2030225.

[35] Florian Lonsing & Uwe Egly (2014): *Incremental QBF Solving*. In: *CP*, *LNCS* 8656, Springer, pp. 514–530, doi:10.1007/978-3-319-10428-7_38.

[36] Kenneth L. McMillan (1993): *The SMV System*. In: *Symbolic Model Checking*, Springer, pp. 61–85, doi:10.1007/978-1-4615-3190-6_4.

[37] Kenneth L McMillan (1999): *The SMV language*. Technical Report, Cadence Berkeley Labs. Available at `http://web.cs.wpi.edu/~kfisler/Courses/525V/S02/Readings/smv-cadence.pdf`.

[38] A. Morgenstern, M. Gesell & K. Schneider (2013): *Solving Games Using Incremental Induction*. In: *IFM'13*, LNCS 7940, Springer, pp. 177–191, doi:10.1007/978-3-642-38613-8_13.

[39] Bryan O'Sullivan: *Attoparsec*. `https://hackage.haskell.org/package/attoparsec`.

[40] Amir Pnueli & Roni Rosner (1989): *On the Synthesis of a Reactive Module*. In: *POPL*, ACM Press, pp. 179–190, doi:10.1145/75277.75293.

[41] Michael O. Rabin (1969): *Decidability of second-order theories and automata on infinite trees*. Trans. Amer. Math. Soc. 141, pp. 1–35, doi:10.1090/S0002-9947-1969-0246760-1.

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

[43] Richard Rudell (1993): *Dynamic variable ordering for ordered binary decision diagrams*. In: *ICCAD*, IEEE Computer Society, pp. 42–47, doi:10.1145/259794.259802.

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

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

[46] Fabio Somenzi (1999): *Binary Decision Diagrams*. In: *Calculational system design*, 173, IOS Press, p. 303.

[47] Wolfgang Thomas (1995): *On the Synthesis of Strategies in Infinite Games*. In: *STACS*, pp. 1–13, doi:10.1007/3-540-59042-0_57.

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