SYNTCOMP – Synthesis Competition for Reactive Systems

Swen Jacobs (TU Graz)
with Roderick Bloem and Rüdiger Ehlers

Thanks to Armin Biere, Timotheus Hell, Ayrat Khalimov, Robert Könighofer, and all participants!

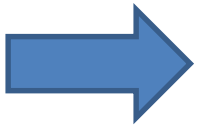
Reactive Synthesis: State of the Art

Currently:

- Tools are not easily comparable
- No standard language
- No comprehensive benchmark set

We want to establish:

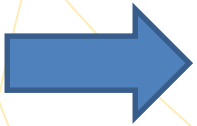
- a **common language** for synthesis problems
- a **repository** for synthesis benchmarks

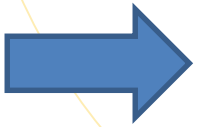


foster research by competition

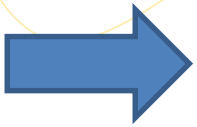
Design Choices

- want a **low entry-barrier** for first competition

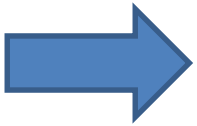
  
  low-level format (no translation from LTL)

  
  restriction to safety properties

- re-use **existing standards**

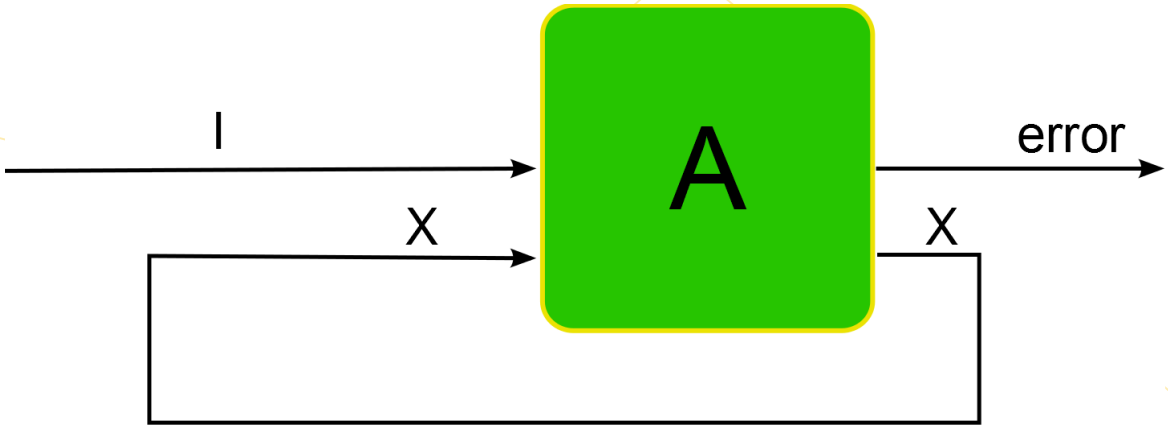
  
  extend well-known AIGER format from HWMCC

- **verifiable correctness**

  
  synthesis output in AIGER format can directly be checked by many model checkers

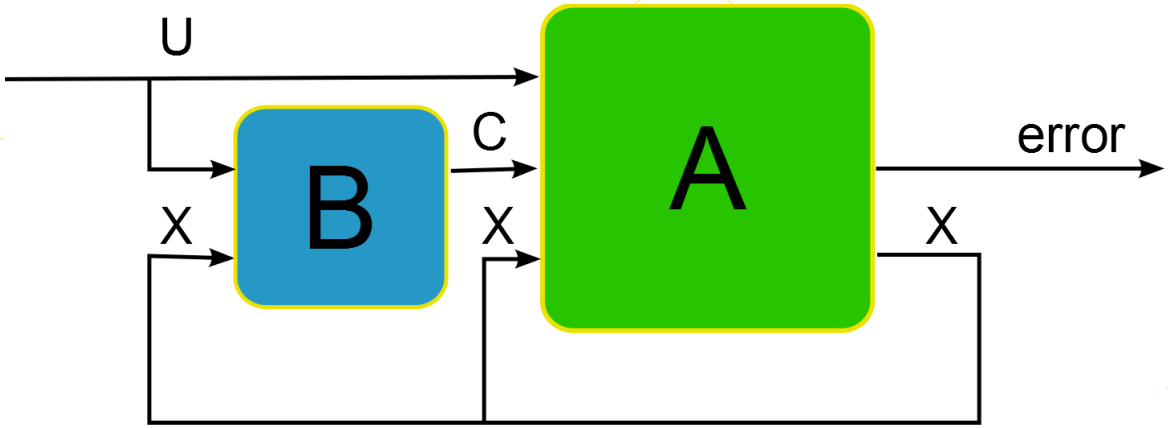
AIGER Format (for model checking)

- AIGER format defines system and spec as a circuit A, composed of And-Gates, Inverters, and Latches
- For safety specs, single output is *error*; system is correct iff *error* is always false



Extended AIGER Format for Synthesis

- For synthesis problems, partition inputs I of system into **controllable** inputs C and **uncontrollable** inputs U
- A solution of synthesis problem is an AIG that includes original AIG A, and adds control structure B for inputs C such that resulting system is correct



Rules

- Realizability Track:
  - determine realizability within time bound
  - **fastest** tool gets most points
- Synthesis Track:
  - find **small** implementation within time bound
    (points distributed according to ranking in size)
  - result must be **verifiable** (within different time bound)
- Sequential and Parallel subtracks in both cases

Rules

157 benchmarks: 1000 / 157 = 6.37 points per benchmark
3 tools solve the problem:

1. 3/6 x 6.37
2. 2/6 x 6.37
3. 1/6 x 6.37

n benchmarks, k tools solve a given problem:

tool with ith best solution receives $\frac{1000}{n} \cdot \frac{k-i+1}{\sum_{j=1}^{k} j}$ points

„best“ is fastest (realizability) or smallest (synthesis)
no solution or not verifiable (synthesis): no points

Benchmark Collection

569 benchmark instances

Classes:

- simple building blocks (adder, counter, bit-shifter, etc.)
- AMBA case study, Generalized Buffer case study (translated from Anzu benchmark set)
- Robotics benchmarks: assembly line, moving obstacles
- translation of Acacia+ benchmark set

Most benchmarks **parameterized** (size, safety translation)

Benchmark Collection

Not in time for 2014 competition:

- driver synthesis (recently translated from Termite specs)
- more robot motion planning (under construction)

Competition Entrants

- tools from 5 different research groups
  - **AbsSynthe** [Raskin Group, UL Bruxelles]
  - **Basil** [Ehlers, Uni/DFKI Bremen]
  - **Demiurge** [Könighofer, Seidl, TU Graz/JKU Linz]
  - **realizer** [Tentrup, Saarland University]
  - **Simple BDD Solver** [Walker, Ryzhyk, NICTA/Uni Toronto]
- tools can compete in ≤3 configurations per track
- overall, 10 (or 19) different solver configurations

AbsSynthe

- Implemented in Python,
  using CUDD BDD package and
  ABC to optimize circuits
- two classical algorithms using BDDs:
  1. classical backward fixpoint computation of an **uncontrollable predecessors** operator.
  2. CEGAR-based algorithm by de Alfaro & Roy [2010] which uses over- and under-approximations of the **UPRE** operator.

AbsSynthe

Optimizations:

- for first algorithm: **partitioned transition relation** to avoid building a single monolithic BDD (cp. Burch, Clarke, Long 1991)
- for second algorithm:
  1. **recycling** information from the over-approximated UPRE operator to
     i) over-approximate the set of reachable states, and
     ii) speed up the computation of the concrete UPRE.
  2. heuristic for chosing latches to make visible to refine an abstraction of the game

(Details in tomorrow‘s talk)

Basil

- Implemented in C++,
  using CUDD BDD package and
  ABC to optimize circuits
- Does **not** use partitioned transition relation

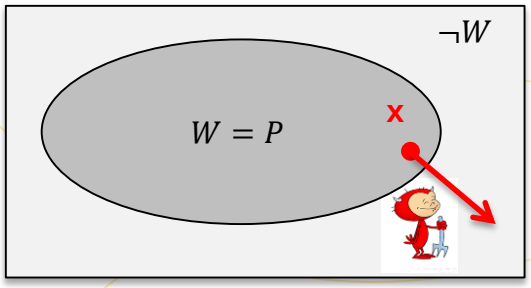
Optimizations:

- BDDs that represent AIG nodes are created in special order
- co-factor-based approach for strategy extraction: compaction of BDD for controllable bits that takes into account reachable state set
  (cp. Bloem et al. [DATE/ENTCS] 2007)

Demiurge

- implemented in C++
- uses SAT-solvers MiniSAT and Lingeling for learning-based approach
- uses QBF solver DepQBF and QBF preprocessor Bloqqer for template-based approach
- uses ABC to optimize circuits

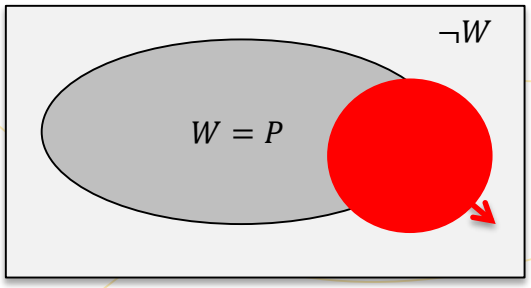
Demiurge: Learning-Based Method



$$\mathbf{x} = x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge x_4$$

- Compute “bad” state using SAT-Solver

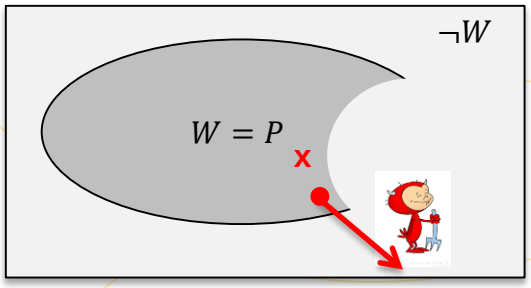
Demiurge: Learning-Based Method



$\mathbf{x} = \quad \neg x_3$

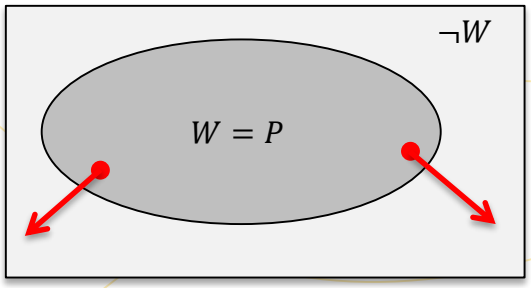
- Compute “bad” state using SAT-Solver
- Generalize it by dropping literals

Demiurge: Learning-Based Method



- Compute “bad” state using SAT-Solver
- Generalize it by dropping literals
- Exclude it from the winning region
- Repeat

Demiurge: Parallelized Method



- Several threads doing this in parallel

Demiurge: Template-Based Method

- $W(\bar{x}, \bar{k})$ is a parameterized winning region
  - Concrete values for $\bar{k}$ → concrete function $W(\bar{x})$
- Solve: $\exists \bar{k} : W(\bar{x}, \bar{k})$ is correct
  - Using a QBF Solver

Demiurge: Circuit Extraction

- Learning-Based Method
  - Using SAT-Solving
- Demiurge as a framework
  - Easy to add new back-ends
- http://www.iaik.tugraz.at/content/research/design_verification/demiurge/

realizer

- implemented in Python
- uses BDD package CUDD
- implements two versions of **UPRE** computation:

1. with **partitioned transition relation** (similar to AbsSynthe)
2. monolithic transition relation, with **optimization** for latches that directly depend on a given variable

- Sequential: only 1st option, Parallel: both

Simple BDD Solver

- implemented in Haskell
- uses BDD package CUDD
- basis of Termite predicate-abstraction solver
- implements **UPRE** computation with:

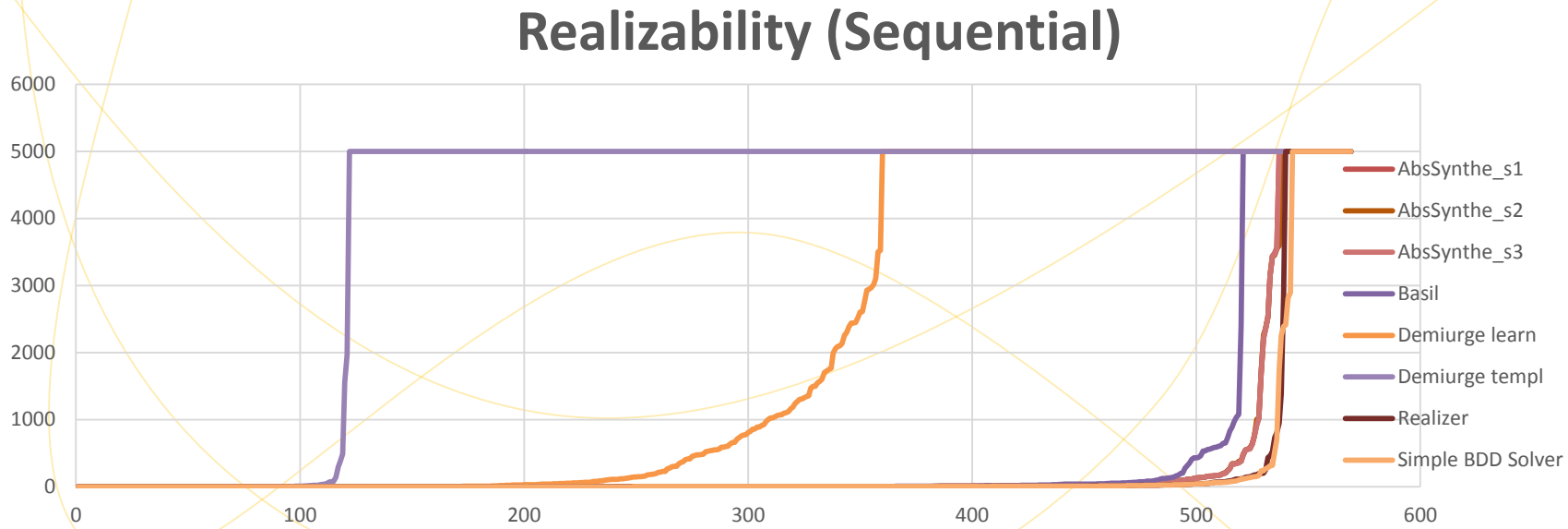
1. **partitioned transition relation**, and
2. BDD reordering by sifting

- no parallel version

Experiment Setup

- EDACC execution & evaluation system
- compute nodes: 2 Octo-Core Intel XEON processors (2.0 GHz), 64 GB RAM
- each job runs isolated on one node
  (what a waste: no tool uses more than 3 cores)
- sequential tracks: 5000s CPU Time
- parallel tracks: 5000s Wall Time
- model checker: iimc

Results



Scores (1000 Pt. divided by 569 benchmarks, distributed over 8 tools):

1. Simple BDD Solver 201
2. Realizer 182
3. Basil 163
   Demiurge 163

Benchmarks Only Solved by One Tool: Realizability (Sequential)

Tool	Benchmark
Basil	factory_assembly_5x5_2_10errors.aag
	factory_assembly_5x5_2_11errors.aag
	factory_assembly_7x5_2_10errors.aag
	factory_assembly_7x5_2_11errors.aag
Realizer	gb_s2_r2_1_UNREAL.aag*
Demiurge (Template)	mult12.aag to mult16.aag
	stay18n.aag
	stay20n.aag

*determined to be realizable

Results

Realizability (Parallel)

Entrants: AbsSynthe, Basil, Demiurge, realizer

Scores (1000 Pt. distributed over 4 tools, 569 benchmarks):

1. Basil 331
2. realizer 279
3. AbsSynthe 219

CPU Time vs. Wall Time

Basil and Demiurge use Wall Time very efficiently.

Smallest CPU Time/ Wall Time for solving any example:

Tool	min CPU Time	min Wall Time
AbsSynthe	0.04	0.36
Basil	0.02	0.05
Demiurge	0.00	0.02
realizer	0.04	0.43

Basil/Demiurge solve ~150/100 problems <0.36s Wall Time

Basil/Demiurge solve ~50/80 problems <0.04s CPU Time

Benchmarks Only Solved by One Tool: Realizability (Parallel)

Tool	Benchmark
AbsSynthe	mult11.aag
	stay16y.aag
Basil	factory_assembly_5x5_2_10errors.aag
	factory_assembly_5x5_2_11errors.aag
	factory_assembly_7x5_2_10errors.aag
	factory_assembly_7x5_2_11errors.aag

Demiurge and **Realizer** in parallel mode (5000s Wall Time) do not solve the problems that have been solved uniquely in sequential mode (5000s CPU Time)

Results

Synthesis (Sequential)

Selection of benchmarks: only realizable benchmarks solved in realizability track (by at least one tool), smaller number of variants of each benchmark.

Entrants: AbsSynthe (x3), Basil, Demiurge (x2)

Scores (1000 Pt. distributed over 6 tools, 157 benchmarks):

1. AbsSynthe (all) 256 (solved+checked* 143)
2. Demiurge (1) 249 (solved+checked 121)
3. Basil 131 (solved+checked* 117)

*not all solutions could be verified

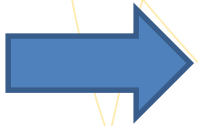
Benchmarks Only Solved by One Tool: Synthesis (Sequential)

No single benchmark only solved by one tool.

Basil and Demiurge could not synthesize solutions for uniquely solved (and realizable) problems from realizability track.

Scoring Issues

In Synthesis, points awarded by size of solution



no problem of CPU Time vs. Wall Time, or very small runtimes.

However, **simple problems** have big weight:

Tool	solved in <0.5s CPU Time	solved in <0.5s Wall Time
AbsSynthe	75	40
Basil	33	27
Demiurge (learn)	54	46

(Total: 157 problems)

The Model Checking Problem

Basil and AbsSynthe produced solutions (~5 each) that **could not be model checked**.

Tools do not get points for these solutions.

Even if these points would have been awarded, ranking would have remained the same.

Results

Synthesis (Parallel)

Selection of benchmarks: same as Synthesis sequential.

Entrants: AbsSynthe, Basil, Demiurge

Scores (1000 Pt. distributed over 3 tools, 157 benchmarks):

1. Demiurge 393 (solved+checked 119)
2. AbsSynthe 352 (solved+checked* 143)
3. Basil 235 (solved+checked* 117)

*not all solutions could be verified

Benchmarks Only Solved by One Tool: Synthesis (Parallel)

Tool	Benchmark
AbsSynthe	amba10c5y.aag
	amba9c5y.aag*
	stay16y.aag

*Basil crashed 3 times with SIGSEGV on this benchmark

Winners – SYNTCOMP 2014

Synthesis (Seq.): **AbsSynthe** [UL Bruxelles]

Synthesis (Par.): **Demiurge** [TU Graz/JKU Linz]

Realizability (Seq.): **Simple BDD Solver** [NICTA/Uni Toronto]

Realizability (Par.): **Basil** [Uni/DFKI Bremen]

Conclusions

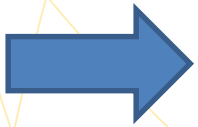
- good number of benchmarks collected
- good number of tools participated
- tools showed different strengths

- **scoring issues**:
  - new tools cannot simply compare scores
  - no „overall scoring“
    (all benchmarks used in realizability,
    only realizable benchmarks in synthesis)
  - CPU Time vs. Wall Time, very small runtimes

- realizability tools did **not profit from parallelism**

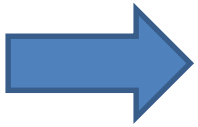
Possible Extensions

- **limitation** to safety



AIGER in principle allows liveness

- many advanced synthesis approaches **impossible or difficult** in low-level format
(smart LTL translations, GR(1), predicate abstraction)



additional track that starts from LTL specs?

- try to ensure that we encourage real progress

Further information

- SYNTCOMP website & blog: www.syntcomp.org
- SYNTCOMP mailinglist: syntcomp@lists.iaik.tugraz.at