Competition Report: CHC-COMP-20

Philipp Rümmer
Uppsala University, Sweden

CHC-COMP-20[1] is the third competition of solvers for Constrained Horn Clauses. In this year, 9 solvers participated at the competition, and were evaluated in four separate tracks on problems in linear integer arithmetic, linear real arithmetic, and arrays. The competition was run in the first week of May 2020 using the StarExec computing cluster. This report gives an overview of the competition design, explains the organisation of the competition, and presents the competition results.

1 Introduction

Constrained Horn Clauses (CHC) have over the last decade emerged as a uniform framework for reasoning about different aspects of software safety [10, 2]. Constrained Horn clauses form a fragment of first-order logic, modulo various background theories, in which models can be constructed effectively with the help of model checking algorithms. Horn clauses can be used as an intermediate verification language that elegantly captures various classes of systems (e.g., sequential code, programs with functions and procedures, concurrent programs, or networks of timed automata) and various verification methodologies (e.g., the use of state invariants, verification with the help of contracts, Owicki-Gries-style invariants, or rely-guarantee methods). Horn solvers can be used as off-the-shelf back-ends in verifiers, and thus enable construction of verification systems in a modular way.

CHC-COMP-20 is the third competition of solvers for Constrained Horn Clauses, a competition affiliated with the 7th Workshop on Horn Clauses for Verification and Synthesis (HCVS) at ETAPS 2020. The goal of CHC-COMP is to compare state-of-the-art tools for Horn solving with respect to performance and effectiveness on realistic, publicly available benchmarks. The deadline for submitting solvers to CHC-COMP-20 was April 30 2020, resulting in 9 solvers participating, which were evaluated in the first week of May 2020. The 9 solvers were evaluated in four separate tracks on problems in linear integer arithmetic, linear real arithmetic, and the theory of arrays. The results of the competition can be found in Section 6 of this report.

Due to the Covid-19 crisis, both ETAPS 2020 and HCVS were postponed, and at the time of finalising this report no new dates had been set; this means that the present report is the main documentation of CHC-COMP-20 at the moment. It is planned, however, that the competition will be presented and discussed in detail also at the HCVS workshop when it takes place, either in physical or virtual form.

1.1 Acknowledgements

CHC-COMP-20 heavily builds on the infrastructure and scripts written for CHC-COMP-18 and CHC-COMP-19, run by Arie Gurfinkel and Grigory Fedyukovich, respectively. Contributors to the competition infrastructure also include Adrien Champion, Dejan Jovanovic, and Nikolaj Bjørner.

Like in the first two competitions, CHC-COMP-20 was run using StarExec [24]. We are extremely grateful for the computing resources and evaluation environment provided by StarExec, and for the fast

[1] `https://chc-comp.github.io/`

Submitted to:
HCVS 2020

and competent support by Aaron Stump and his team whenever problems occurred. CHC-COMP-20 would not have been possible without this!

The organiser of CHC-COMP-20 is supported by the Swedish Research Council (VR) under grant 2018-04727, and by the Swedish Foundation for Strategic Research (SSF) under the project WebSec (Ref. RIT17-0011).

2 Brief Overview of the Competition Design

2.1 Competition Tracks

In CHC-COMP-20 the same tracks as in CHC-COMP-19 were used for evaluating solvers:

- **LIA-nonlin**: benchmarks with at least one non-linear clause, and linear integer arithmetic as background theory;
- **LIA-lin**: benchmarks with only linear clauses, and linear integer arithmetic as background theory;
- **LIA-lin-arrays**: benchmarks with only linear clauses, and the combined theory of linear integer arithmetic and arrays as background theory;
- **LRA-TS**: benchmarks encoding transition systems, with linear real arithmetic as background theory. Benchmarks in this track have exactly one uninterpreted relation symbol, and exactly three linear clauses encoding initial states, transitions, and error states.

2.2 Computing Nodes

The competition was run on 30 nodes provided by StarExec. Each node had two quadcore CPUs, and each node was used to run two jobs in parallel during the competition runs. The machine specs are:

```
Intel(R) Xeon(R) CPU E5-2609 0 @ 2.40GHz (2393 MHZ)
    10240  KB Cache
    263932744 kB main memory

# Software:
OS:       CentOS Linux release 7.7.1908 (Core)
kernel:   3.10.0-1062.4.3.el7.x86_64
glibc:    glibc-2.17-292.el7.x86_64
          gcc-4.8.5-39.el7.x86_64
          glibc-2.17-292.el7.i686
```

2.3 Test and Competition Runs

All solvers submitted to CHC-COMP-20 were evaluated twice:

- in a first set of **test runs**, in which pre-submissions of the solvers were evaluated to check their configurations and identify possible inconsistencies. For the test runs a smaller set of randomly selected benchmarks was used. The results of the test runs for each solver were directly communicated to the team submitting the solver, but not made public and not shared with other teams. In the test runs, each solver-benchmark pair was limited to 600s CPU time, 600s wall-clock time, and 64GB memory.

- in the **competition runs**, the results of which determined the outcome of CHC-COMP-20. The selection of the benchmarks for the competition runs is described in Section 4, and the evaluation of the competition runs in Section 2.4. In the competition runs, each job was limited to 1800s CPU time, 1800s wall-clock time, and 64GB memory.

2.4 Evaluation of the Competition Runs

The ranking of solvers in each track was done based on the **Score** reached by the solvers in the competition run for that track. In case two solvers had equal **Score**, the ranking of the two solvers was determined by **CPU time**. It was assumed that the outcome of running one solver on one benchmark can only be **sat**, **unsat**, or **unknown**; the last outcome includes solvers giving up, running out of resources, or crashing.

The definition of **Score** and **CPU time** are:

- **Score**: the number of **sat** or **unsat** results produced by a solver on the benchmarks of a track.
- **CPU time**: the average CPU time needed by a solver to produce the result **sat** or **unsat** in some track, not counting the runtime for **unknown** results. This average time was computed based on the CPU time stamp reported by StarExec for the output of the result `sat`/`unsat`.

In addition, the following features were computed for each solver and each track:

- **Wall-clock time**: the average wall-clock time needed by a solver to produce the result **sat** or **unsat** in some track, not counting the runtime for **unknown** results.
- **Speedup**: the ratio **CPU time / Wall-clock time**.
- **SotAC**: the state-of-the-art contribution of a solver, computed in the same way as in the CADE ATP System Competition (CASC). The **SotAC** of a benchmark in some track is the inverse of the number of systems that reported **sat** or **unsat** for the benchmark. The **SotAC** of a solver is the average **SotAC** of the benchmarks it could solve.

3 Competition Benchmarks

3.1 File Format

CHC-COMP represents benchmarks in a fragment of the SMT-LIB 2.6 format. The fragment is defined on `https://chc-comp.github.io/format.html`. For CHC-COMP-20, several minor modifications were done in the format definition, in particular the role of nullary predicates was clarified. The conformance of a well-typed SMT-LIB script with the CHC-COMP fragment can be checked using the `format-checker` available on `https://github.com/chc-comp/chc-tools`.

3.2 Benchmark Processing

All benchmarks used in CHC-COMP-20 were pre-processed using the `format.py` script available in the repository `https://github.com/chc-comp/scripts`, using the command line

```
> python3 format.py --out_dir <outdir> --merge_queries True <smt-file>
```

The script tries to translate arbitrary Horn-like problems in SMT-LIB format to problems within the CHC-COMP fragment. Only benchmarks processed in this way were used in the competition. The option `--merge_queries` was added for CHC-COMP-20 to the script, and has the effect of merging

multiple queries in a benchmark into a single query by introducing an auxiliary nullary predicate. For a discussing on this, see Section 3.3.

After processing with `format.py`, benchmarks were checked and categorised into the four tracks using the `format-checker` scripts available on `https://github.com/chc-comp/chc-tools`.

Benchmarks that could not be processed by `format.py`, were rejected by the `format-checker`, or did not conform to any of the competition tracks, were not used in CHC-COMP-20.

3.3 Handling of Benchmarks with Multiple Queries

In the CHC-COMP format, a query is a clause with the head `false`, and encodes the property to be verified in a benchmark. At the moment, the CHC-COMP fragment of SMT-LIB only allows benchmarks with exactly one query, which means that problems involving multiple queries have to be mapped to the single-query format. Of the CHC-COMP benchmarks, a significant number contains multiple queries.

In the past competitions, this was handled by splitting benchmarks with multiple queries into multiple single-query benchmarks (option `--split_queries` of `format.py`); in CHC-COMP-20, instead benchmarks with multiple queries were converted to the single-query format by introducing an auxiliary nullary predicate (or Boolean variable), and merging all queries to a single query (`--merge_queries` of `format.py`). The motivation for the new pre-processing is that splitting of queries sometimes makes benchmarks artificially hard: with multiple queries, often some of the queries can be disproven easily, while other queries are difficult to prove or disprove. Such problems, seen as a whole, are simple to solve, but splitting and considering all queries individually will lead to some hard benchmarks. Since the benchmarks used in CHC-COMP should represent realistic applications, this is an undesired effect.

In future editions of CHC-COMP, it might be even better to just allow benchmarks with multiple queries, and extend the CHC-COMP format accordingly. Many solvers are already now able to process benchmarks with multiple queries. Other solvers could decide themselves whether benchmarks with multiple queries should be handled by splitting or by merging; both translations are quite simple to implement in a solver, or alternatively a solver could invoke the existing script for pre-processing.

3.4 Benchmark Inventory

In contrast to most other competitions, CHC-COMP stores benchmarks in a decentralised way, in multiple repositories managed by the contributors of the benchmarks themselves. Table 1 summarises the number of benchmarks that were obtained by collecting benchmarks from all available repositories using the process in Section 3.2. Duplicate benchmarks were identified by computing a checksum for each (processed) benchmark, and were discarded.

The repository chc-comp19-benchmarks of benchmarks selected for CHC-COMP-19 was included in the collection, because this repository contains several unique families of benchmarks that are not available in other repositories under `https://github.com/chc-comp`. Such benchmarks include problems generated by the Ultimate tools in the LIA-lin-arrays track.

From jayhorn-benchmarks, only the problems generated for sv-comp-2020 were considered, which subsume the problems for sv-comp-2019.

4 Benchmark Rating and Selection

This section describes how the benchmarks for CHC-COMP-20 were selected among the unique benchmarks summarised in Table 1. For the competition tracks LIA-lin-arrays and LRA-TS, the benchmark

Repository	LIA-nonlin	LIA-lin	LIA-lin-arrays	LRA-TS
eldarica-misc	69 / 66	147 / 134		
extra-small-lia		55 / 55		
hcai-bench	135 / 133	100 / 86	39 / 39	
hopv	68 / 67	49 / 48		
jayhorn-benchmarks	5138 / 5084	75 / 73		
kind2-chc-benchmarks	851 / 738			
ldv-ant-med			10 / 10	
llreve-bench	59 / 57	44 / 44	31 / 31	
quic3			43 / 43	
sally-chc-benchmarks				177 / 174
seahorn	72 / 70	3421 / 2847		
tricera-benchmarks	4 / 4	405 / 405		
vmt-chc-benchmarks		906 / 803		99 / 98
sv-comp	1643 / 1169	3150 / 2932	79 / 73	
chc-comp19-benchmarks	271 / 265	326 / 314	305 / 305	229 / 227
Total	8310 / 7653	8678 / 7741	507 / 501	505 / 499

Table 1: Summary of benchmarks available on `https://github.com/chc-comp` and in the StarExec CHC space. For each collection of benchmarks and each CHC-COMP-20 track, the first number gives the total number of benchmarks, and the second number the number of contributed unique benchmarks (after discarding duplicate benchmarks).

library only contains 501 and 499 unique benchmarks, respectively, which are small enough sets to use all benchmarks in the competition. For the tracks LIA-nonlin and LIA-lin, in contrast, too many benchmarks are available, so that a representative sample of the benchmarks had to be chosen.

To gauge the difficulty of the available problems in LIA-nonlin and LIA-lin, a simple rating based on the performance of the CHC-COMP-19 solvers was computed. For this, the two top-ranked competing solvers from CHC-COMP-19 were run for a few seconds on each of the LIA-nonlin and LIA-lin benchmarks:

- **Eldarica:** was run with a 5s timeout on each benchmark.[2] Eldarica runs entirely in managed code on a JVM; to avoid frequent JVM restarts, the daemon mode of Eldarica was used. Otherwise, the same binary and same options were chosen as in CHC-COMP-19.
- **Ultimate Unihorn Automizer:** was run with a slightly higher timeout of 8s, since the solver also partly runs in managed code, but (to the best of the organiser's knowledge) does not have a similar daemon mode as Eldarica. The same binary and options as in CHC-COMP-19 were used.

The outcomes of those test runs gave rise to three possible ratings for each benchmark:

- **A:** both tools were able to determine the benchmark status within the given time budget.
- **B:** only one tool could determine the benchmark status.
- **C:** both tools timed out.

[2]Run on an Intel Core i5-650 2-core machine with 3.2GHz.

	LIA-nonlin			LIA-lin		
Repository	#A /	#B /	#C	#A /	#B /	#C
eldarica-misc	12 /	28 /	26	26 /	91 /	17
extra-small-lia				3 /	24 /	28
hcai-bench	19 /	71 /	43	59 /	19 /	8
hopv	26 /	38 /	3	45 /	2 /	1
jayhorn-benchmarks	49 /	2680 /	2355	55 /	18 /	
kind2-chc-benchmarks	58 /	179 /	501			
llreve-bench	6 /	35 /	16	9 /	35 /	
seahorn	6 /	34 /	30	753 /	323 /	1771
tricera-benchmarks	1 /	3 /		9 /	23 /	373
vmt-chc-benchmarks				33 /	252 /	518
sv-comp	25 /	1057 /	87	968 /	1855 /	109
chc-comp19-benchmarks	42 /	116 /	107	31 /	100 /	183
Total	244 /	4241 /	3168	1991 /	2742 /	3008

Table 2: The number of unique LIA-nonlin and LIA-lin benchmarks with ratings A / B / C.

The number of benchmarks per rating are shown in Table 2. As can be seen from the table, the simple rating method separates the benchmarks into partitions of comparable size, and provides some information about the relative hardness of the problems in the different repositories.

From each repository r, up to $3 \cdot N_r$ benchmarks were then selected randomly: N_r benchmarks with rating A, N_r benchmarks with rating B, and N_r benchmarks with rating C. If a repository contained fewer than N_r benchmarks for some particular rating, instead benchmarks with the next-higher rating were chosen. As special cases, up to N_r benchmarks were selected from repositories with only A-rated benchmarks; up to $2 \cdot N_r$ benchmarks from repositories with only B-rated benchmarks; and up to $3 \cdot N_r$ benchmarks from repositories with only C-rated benchmarks.

The number N_r was chosen individually for each repository, based on a manual inspection of the repository to judge the diversity of the contained benchmarks. The chosen N_r, and the numbers of selected benchmarks for each repository, are given in Table 3.

For the actual selection of benchmarks with rating X, the following Unix command was used:

```
> cat <rating-X-benchmark-list> | sort -R | head -n <num>
```

The final set of benchmarks selected for CHC-COMP-20 can be found in the github repository `https://github.com/chc-comp/chc-comp20-benchmarks`, and on StarExec in the public space `CHC/CHC-COMP/chc-comp20-benchmarks`.

5 Solvers Entering CHC-COMP-20

In total, 9 solvers were submitted to CHC-COMP-20: 8 competing solvers, and one further solver (Eldarica, co-developed by the competition organiser) that was entering outside of the competition. A summary of the participating solvers is given in Table 4.

More details about the participating solvers are provided in the solver descriptions in Section 8. The binaries of the solvers used for the competition runs can be found in the public StarExec space `CHC/CHC-COMP/chc-comp20-benchmarks`.

	LIA-nonlin		LIA-lin		LIA-lin-arrays	LRA-TS
Repository	N_r	#Sel	N_r	#Sel	#Selected	#Selected
eldarica-misc	10	30	15	45		
extra-small-lia			10	30		
hcai-bench	20	60	15	38	39	
hopv	10	23	10	13		
jayhorn-benchmarks	30	90	10	20		
kind2-chc-benchmarks	30	90				
ldv-ant-med					10	
llreve-bench	15	45	15	30	31	
quic3					43	
sally-chc-benchmarks						174
seahorn	15	45	30	90		
tricera-benchmarks	1	2	20	60		
vmt-chc-benchmarks			30	90		98
sv-comp	30	90	30	90	73	
chc-comp19-benchmarks	30	90	30	90	305	227
Total		565		596	501	499

Table 3: The number of selected unique benchmarks for the four CHC-COMP-20 tracks.

6 Competition Results

6.1 Overview

The winners and top-ranked solvers of the four CHC-COMP-20 tracks are:

	LIA-nonlin	LIA-lin	LIA-lin-arrays	LRA-TS
Winner	**Spacer**	**Spacer**	**Spacer**	**IC3IA**
Place 2	Eldarica-abs	Eldarica-abs	Ultimate Unihorn Automizer	Sally (two config.)
Place 3	Ultimate Unihorn Automizer	Ultimate Unihorn Automizer	ProphIC3	Spacer

6.2 Detailed Results

Detailed results for the four tracks can be found in Figures 1, 2, 3, and 4.

6.3 Observed Inconsistencies in the Competition Runs, and Fixes

LIA-lin-arrays: Only one case of inconsistent results was observed in the competition runs, namely the results for benchmark `chc-LIA-lin-arrays_381.smt2` in the LIA-lin-arrays track. Ultimate Unihorn Automizer claimed that this benchmark is satisfiable, whereas Spacer reported that it is unsatisfiable; all other solvers timed out.

This issue was discussed with the authors of the solvers. Spacer can produce an internal counterexample for the problem, but no counterexample for the complete input problem that could be verified

Solver	LIA-nonlin	LIA-lin	LIA-lin-arrays	LRA-TS
Eldarica-abs	`def`	`def`	—	—
IC3IA	—	`default.sh`	`default.sh`	`default.sh`
PCSat	`pcsat_dt_tb`	`pcsat_dt_tb`	—	—
ProphIC3	—	—	`default.sh`	—
Sally	—	—	—	`y2o2_-decomposing_-itp, parallel`
Spacer	`lia`	`lia_lin`	`arrays`	`lra`
Ultimate TreeAutomizer	`default`	`default`	`default`	`default`
Ultimate Unihorn Automizer	`default`	`default`	`default`	`default`
Eldarica (Hors Concours)	`def`	`def`	`def`	—

Table 4: The submitted solvers, and the configurations used in the individual tracks.

by independent tools. Ultimate Unihorn Automizer does not have functionality to output models. No obvious bug was found in either of the tools. This means that there is no immediate way to establish the actual status of the benchmark beyond doubt, and no way to tell which of the solvers gave the right answer; but clearly one of the tools contains a bug.

Since no further inconsistencies were observed in the competition, in this particular case the organiser decided to remove the benchmark `chc-LIA-lin-arrays_381.smt2` and not count Spacer's nor Ultimate Unihorn Automizer's answer. The issue highlights a problem in the competition design, however, which should be addressed in the next competition (see Section 7 for more thoughts about this).

Fixes in IC3IA and ProphIC3: After the submission deadline, the team submitting the solvers IC3IA and ProphIC3 reported that they had found a bug affecting the soundness of both tools in the LIA-lin-arrays track, and provided new versions of the tools. No inconsistencies were observed for either tool in the competition runs, but since there was enough time the experiments in LIA-lin-arrays were repeated with the new versions of the solvers. This changed the results in LIA-lin-arrays in the following way:

	#sat	#unsat
IC3IA (original)	96	47
IC3IA (fixed)	92	55
ProphIC3 (original)	183	74
ProphIC3 (fixed)	140	74

The performance of IC3IA increased slightly, and the new version reached the same score as Ultimate TreeAutomizer. The performance of ProphIC3, in contrast, decreased significantly, and ProphIC3 moved from the second to the third place.

6.4 Resource Budgets: CPU time vs. Wall-clock time

In the preparation phase of CHC-COMP-20, there was a discussion with several teams about the use of CPU time vs. wall-clock time to define the resource budget of solvers, and the possibility to have separate tracks in future editions of CHC-COMP for parallel solvers (with wall-clock time budget and no limit on CPU time). For CHC-COMP-20, such tracks were not introduced in the end, and the primary resource limit was the available CPU time, but adding tracks for parallel solvers can be an interesting new feature for CHC-COMP-21, or beyond.

It is meaningful to analyse the CHC-COMP-20 outcomes in this light. An immediate observation is that the three LIA tracks show very different behaviour with respect to runtime than the LRA-TS track. In the LIA tracks, the average CPU time of solvers is usually significantly below 100s, and the cactus plots show that the ranking of solvers hardly changes above 100s CPU time (Figures 1, 2, 3). Only few benchmarks in those tracks can be solved with CPU time above 600s. This means that CPU time (and therefore also wall-clock time) is essentially not a limiting factor in the LIA tracks, the current solvers already hardly utilise the available computation time. Since there is plenty of computation time available, already at this point solvers can use portfolios and run different configurations in parallel, as done by some of the participating solvers. To make specific parallel LIA tracks interesting, it would be necessary to change the scoring scheme of CHC-COMP: no longer count just the number of solved benchmarks, but also factor in the required wall-clock time in the ranking of solvers.

The situation is different in LRA-TS: in this track the average CPU time used by solvers is between 100s and 300s, and the cactus plots show interesting developments even after 600s CPU time. Since the maximum speed-up observed in LRA-TS is 2.93, the cactus plot for wall-clock performance should be interpreted only up to a time limit of around 600s, beyond 600s the results are limited by the available CPU time. Limiting the wall-clock time to 600s would indeed change the ranking of solvers, with the parallel computation used in pSally paying off, and other solvers could of course be optimised in a similar way. It would be an interesting experiment to repeat the evaluation of LRA-TS with unlimited CPU time, and wall-clock time limited to 1800s, which would amount to a parallel track.

In summary, for the next editions of CHC-COMP parallel tracks are mainly interesting for LRA-TS, where solvers can indeed utilise the available computation time. In the LIA tracks the solvers are mainly limited by the implemented algorithms and heuristics, and less by the available computation time.

7 Conclusions

The organiser would like to congratulate the winners of the four CHC-COMP-20 tracks, Spacer and IC3IA, and all solvers and people submitting solvers for the excellent performance this year! Thanks go to all people who have been helping with infrastructure, scripts, benchmarks, or in other ways, see the acknowledgements in the introduction; and to the HCVS workshop for hosting CHC-COMP!

In order to keep CHC-COMP an interesting and relevant competition, the organiser also identified several questions and issues that should be discussed and addressed in the next editions:

- Duplicate benchmarks included in multiple repositories. The selection of benchmarks in CHC-COMP-20 probably contained some benchmarks repeatedly, for instance benchmarks from the repositories sv-comp or chc-comp19-benchmarks that were pre-processed using different tools initially, and thus not identified as duplicates. The outcome of the competition was probably not affected very much by this, but this is an issue that should be addressed in the long run. To keep

the decentralised model of storing benchmarks in CHC-COMP, one could maintain a global list of all available unique benchmarks.

- A more systematic method for hardness rating of benchmarks is needed. Some rating is required to select interesting benchmarks, but any rating has the potential of changing the outcome of the competition. The use of Eldarica and Ultimate Unihorn Automizer (in the versions from CHC-COMP-19) for problem rating possibly puts the CHC-COMP-20 versions of those solvers at a disadvantage (or advantage) compared to other solvers not used for rating. For instance, if solvers A and B show completely uncorrelated performance on some set of benchmarks, then picking 50% easy and 50% hard benchmarks for solver A will not affect the expected outcome for solver B, but will fix the outcome of solver A to 50%.
- A better way has to be found to handle cases of inconsistent results, as observed this year in the LIA-lin-arrays track. There are different solutions: (i) only use benchmarks with known status in the competition; (ii) require solvers to produce, on demand, models or counterexamples when they claim to be able to solve a problem. This requires a discussion.
- An approach to determine and store the expected outcome of the individual benchmarks.
- A discussion is needed about new tracks to be added in the competition: parallel tracks, or tracks with new background theories, for instance algebraic data-types or bit-vectors?
- **A bigger set of benchmarks is needed, and all users and tool authors are encouraged to submit benchmarks!** In particular, in the LIA-nonlin and LRA-TS tracks, the competition results indicate that harder benchmarks are required.

8 Solver Descriptions

The tool descriptions in this section were contributed by the tool submitters, and the copyright on the texts remains with the individual authors.

Eldarica-abs

Xiaozhen Zhang
Dalian University of Technology, China

Weiqiang Kong
Dalian University of Technology, China

Algorithm. Eldarica-abs is a variant of Eldarica [15], which is a model checker for Horn clauses, Numerical Transition Systems, and software programs. Eldarica-abs mainly concentrates on the modification of the exploration strategies of abstraction lattices utilized in Eldarica. Identical with Eldarica, the inputs of Eldarica-abs can be read in a variety of formats, including SMT-LIB v2 and Prolog for Horn clauses, and fragments of Scala and C for software programs; and these inputs can be analysed using a variant of the Counterexample-Guided Abstraction Refinement (CEGAR) method, in which interpolation-based techniques are used to synthesis new predicates for CEGAR. Different from Eldarica, Eldarica-abs takes the cost-guided bidirectional heuristic search strategy to search constructed abstraction lattices in order to compute the interpolants of good quality [25]. As the goal elements are not only maximal and feasible, but also of minimal cost, this search strategy chooses from the top of the abstraction lattice to search downwards and introduces the predecessors computing function to avoid the repeated visit of the predecessors; and then, for the feasible predecessors of one element under consideration, upward searching strategy is taken to search an element whose all successors are all infeasible; for the infeasible predecessors, the cost and infeasibility information contained in these elements are utilized to change the set of the elements to be searched; additionally, for the selection of every element to be extended, we make the most of the information contained in the visited elements set and the set of elements to be visited, in order to choose the best candidate element and then promote the performance of the searching process along an effective path.

Architecture and Implementation. The overall architecture of Eldarica-abs is the same as Eldarica and it can be divided into three phases: input encoding phase; preprocessing phase; CEGAR engine solving phase [15]. Among these phases, interpolation abstraction is introduced as an effective convergence heuristic technique to compute the Craig interpolants of good quality, which are utilized to inspire the acquirement of the right predicates necessary in CEGAR process. In the course of obtaining suitable Craig interpolants, Eldarica-abs takes a different cost-guided bidirectional heuristic exploration strategy to search the constructed abstraction lattices.

Based on the work of Eldarica, Eldarica-abs is also implemented in Scala and depends on Java and Scala libraries and the Princess SMT solver.

Configuration in CHC-COMP-20. Eldarica-abs was running with default options in the competition.

`https://github.com/zhangxiaozhen/Eldarica-abs`
BSD licence

IC3IA 2020.05

Alberto Griggio
Fondazione Bruno Kessler, Italy

Ahmed Irfan
Stanford University, USA

Makai Mann
Stanford University, USA

Algorithm. The tool is an open-source implementation of IC3 Modulo Theories via Implicit Predicate Abstraction (IC3IA). It is one approach for extending IC3 to the theory level, with the advantage that it can be applied to arbitrary theories without theory-specific quantifier elimination procedures.

Architecture and Implementation. It depends on MathSAT 5.6.3 [7]. The **ic3ia** code is distributed under the GPLv3 license. This tool operates on transition systems rather than CHC. CHC clauses are translated to Verification Modulo Theories (VMT) format using a program distributed with **ic3ia**.

Configuration in CHC-COMP-20. The submitted solver is run using options `-solver-approx 1 -inc-ref 1`.

`https://es-static.fbk.eu/people/griggio/ic3ia/index.html`
GPLv3

PCSat

Yu Gu
University of Tsukuba, Japan

Hiroshi Unno
University of Tsukuba, Japan

Algorithm. PCSat is a solver for a general class of second-order constraints on predicate and function variables. Its applications include but not limited to branching-time temporal verification, dependent refinement type inference, program synthesis, and infinite-state game solving.

PCSat is based on CounterExample-Guided Inductive Synthesis (CEGIS), with the support of multiple synthesis engines including template-based, decision-tree-based, and graphical-model-based [23] ones.

Architecture and Implementation. PCSat is designed and implemented as a highly-configurable solver, allowing us to test various possible combinations of synthesis engines, example sampling strategies, and backend SAT/SMT solvers. This design is enabled by the powerful module system and the metaprogramming features of the OCaml functional programming language.

Configuration in CHC-COMP-20. We adopted a parallel combination of the template-based and decision-tree-based synthesis engines. Z3 and MiniSat are used as the backend SMT and SAT solvers, respectively.

`https://github.com/hiroshi-unno/coar`
Apache License 2.0

ProphIC3

Makai Mann
Stanford University, USA

Ahmed Irfan
Stanford University, USA

Alberto Griggio
Fondazione Bruno Kessler, Italy

Oded Padon
Stanford University, USA

Clark Barrett
Stanford University, USA

Algorithm. The tool **prophic3** is a prototype implementation of a Counter-example Guided Abstraction Refinement (CEGAR) [20] algorithm for model checking modulo the theory of arrays. The algorithm abstracts the theory of arrays using uninterpreted functions and lazily adds array axioms. Furthermore, it uses counterexamples to add history and prophecy variables which can help find simpler invariants, even reducing quantified invariants to quantifier-free invariants in some cases. The approach wraps a standard model checker. The underlying model checker must support all the theories used in the input problem, except the theory of arrays.

Architecture and Implementation. The **prophic3** prototype depends on **ic3ia** [11], an implementation of IC3 Modulo Theories via Implicit Predicate Abstraction (IC3IA) [6]. IC3IA is one approach for extending IC3 [5] to arbitrary theories. This is version 2020.05 of **ic3ia**. It depends on MathSAT 5.6.3 [7]. This tool operates on transition systems rather than CHC. CHC clauses are translated to Verification Modulo Theories (VMT) format using a program distributed with **ic3ia**.

Configuration in CHC-COMP-20. The submitted solver is a portfolio approach which runs **prophic3** from git tag `chccomp-2020` with option `-no-eq-uf`, as well as bounded model checking [1] up to bound 100 on the concrete system.

`https://github.com/makaimann/prophic3`
GPLv3

Sally and pSally

Martin Blicha[3]
Università della Svizzera italiana, Switzerland

Algorithm. Our competition entry is derived from the model checker Sally [17], which we have enhanced in two ways: First, we have supplied our interpolating SMT solver OpenSMT with a specialized interpolation procedure for Linear Real Arithmetic (LRA) as part of Sally's backend. Secondly, we have implemented a parallel version of Sally (pSally) where multiple instances cooperate by sharing information discovered about the problem at hand. The current version of the tool is limited to solving safety of Transition Systems encoded in LRA.

The tool uses Sally's PD-KIND engine [17] as the core algorithm. PD-KIND strengthens the IC3 algorithm with k-induction and gradually builds a k-inductive safe invariant of the transition system. It relies on an SMT solver for answering satisfiability queries, generalization and interpolation. Our specialized interpolation procedure computes stronger interpolants than traditional interpolation algorithm

[3]This submitted tool builds on the work of Dejan Jovanović and Bruno Dutertre. The tool and the related research were realised with significant contributions by various colleagues, in particular by Matteo Marescotti, Antti E. J. Hyvärinen and Natasha Sharygina.

and this has been shown to be useful in model checking scenarios [3]. The parallel version on the other hand leverages a portfolio of interpolation algorithms for discovering useful facts about the system under analysis, as well as a cooperative framework for sharing the discovered information [4].

Architecture and Implementation. The competition entry uses Sally's PD-KIND reasoning engine, with the SMT solvers Yices2 [9] and OpenSMT [16] for generalization and interpolation, respectively. OpenSMT uses an interpolation algorithm that computes *decomposed* LRA interpolants [3]. The parallel version uses SMTS framework [21] for managing multiple instances and their communication.

Configuration in CHC-COMP-20.
Command line options of Sally:

```
$ sally --engine pdkind --solver y2o2 --solver-logic QF_LRA
--pdkind-minimize-frames --pdkind-minimize-interpolants
--opensmt2-itp-lra 4 --opensmt2-simplify_itp 4 --yices2-mode dpllt --output-lang
horn -i <input_file>
```

Command line options of pSally for creating three instances and enabling the sharing of information:

```
$ python3 smts.py -l -s3
```

`http://sri-csl.github.io/sally/`
GNU GENERAL PUBLIC LICENSE v2

`https://zenodo.org/record/3484097`
MIT LICENSE

SPACER

Hari Govind V K
University of Waterloo, Canada

Arie Gurfinkel
University of Waterloo, Canada

Algorithm. SPACER [19] is an IC3/PDR-style algorithm for solving linear and non linear CHCs. Given a set of CHCs, it iteratively proves the unreachability of *false* at larger and larger depths until a model is found or the set of CHCs is proven unsatisfiable. To prove unreachability at a particular depth, SPACER recursively generates sets of predecessor states (called proof obligations (POBs)) from which *false* can be derived and blocks them. Once a POB is blocked, SPACER generalizes the proof to learn a *lemma* that blocks multiple POBs. SPACER uses many heuristics to learn lemmas. These include interpolation, inductive generalization and quantifier generalization. The latest version of Spacer presents a new heuristic for learning lemmas [18].

The current implementation of SPACER supports linear and nonlinear CHCs in the theory of Arrays, Linear Arithmetic and FixedSizeBitVectors. SPACER can generate both quantified and quantifier free models as well as resolution proof of unsatisfiability.

Architecture and Implementation. SPACER is implemented on top of the Z3 theorem prover. It uses many SMT solvers implemented in Z3. Additionally, it implements a max-SMT solver and an interpolating SMT solver.

Configuration in CHC-COMP-20. We used different configurations for different tracks. However, all the preprocessing options and some SPACER options remained the same in all the tracks. Common configuration for all tracks:

```
fp.spacer.global=true fp.spacer.concretize=true fp.spacer.conjecture=true
fp.xform.tail_simplifier_pve=false fp.validate=true fp.spacer.mbqi=false
```

Additionally, in the Arrays track, we used the quantifier generalization strategies:

```
fp.spacer.q3.use_qgen=true fp.spacer.q3.instantiate=true fp.spacer.q3=true
```

In the LRA-TS track, we turned off interpolation:

```
fp.spacer.use_iuc=false
```

`https://github.com/Z3Prover/z3`
MIT License

Ultimate TreeAutomizer 0.1.25-6b0a1c7

Matthias Heizmann
University of Freiburg, Germany

Daniel Dietsch
University of Freiburg, Germany

Jochen Hoenicke
University of Freiburg, Germany

Alexander Nutz
University of Freiburg, Germany

Andreas Podelski
University of Freiburg, Germany

Algorithm. The ULTIMATE TREEAUTOMIZER solver implements an approach that is based on tree automata [8]. In this approach potential counterexamples to satisfiability are considered as a regular set of trees. In an iterative CEGAR loop we analyze potential counterexamples. Real counterexamples lead to an *unsat* result. Spurious counterexamples are generalized to a regular set of spurious counterexamples and subtracted from the set of potential counterexamples that have to be considered. In case we detected that all potential counterexamples are spurious, the result is *sat*. The generalization above is based on tree interpolation and regular sets of trees are represented as tree automata.

Architecture and Implementation. TREEAUTOMIZER is a toolchain in the ULTIMATE framework. This toolchain first parses the CHC input and then runs the `treeautomizer` plugin which implements the above mentioned algorithm. We obtain tree interpolants from the SMT solver SMTInterpol[4] [14]. For checking satisfiability, we use the Z3 SMT solver[5]. The tree automata are implemented in ULTIMATE's automata library[6]. The ULTIMATE framework is written in Java and build upon the Eclipse Rich Client Platform (RCP). The source code is available at GitHub[7].

[4] `https://ultimate.informatik.uni-freiburg.de/smtinterpol/`
[5] `https://github.com/Z3Prover/z3`
[6] `https://ultimate.informatik.uni-freiburg.de/automata_library`
[7] `https://github.com/ultimate-pa/`

Configuration in CHC-COMP-20. Our StarExec archive for the competition is shipped with the `bin/starexec_run_default` shell script calls the ULTIMATE command line interface with the `TreeAutomizer.xml` toolchain file and the `TreeAutomizerHopcroftMinimization.epf` settings file. Both files can be found in toolchain (resp. settings) folder of ULTIMATE's repository.

`https://ultimate.informatik.uni-freiburg.de/`
LGPLv3 with a linking exception for Eclipse RCP

Ultimate Unihorn 0.1.25-6b0a1c7

Matthias Heizmann
University of Freiburg, Germany

Daniel Dietsch
University of Freiburg, Germany

Jochen Hoenicke
University of Freiburg, Germany

Alexander Nutz
University of Freiburg, Germany

Andreas Podelski
University of Freiburg, Germany

Algorithm. ULTIMATE UNIHORN reduces the satisfiability problem for a set of constraint Horn clauses to a software verfication problem. In a first step UNIHORN applies a yet unpublished translation in which the constraint Horn clauses are translated into a recursive program that is nondeterministic and whose correctness is specified by an assert statement The program is correct (i.e., no execution violates the assert statement) if and only if the set of CHCs is satisfiable. For checking whether the recursive program satisfies its specification, Unihorn uses ULTIMATE AUTOMIZER [12] which implements an automata-based approach to software verification [13].

Architecture and Implementation. ULTIMATE UNIHORN is a toolchain in the ULTIMATE framework. This toolchain first parses the CHC input and then runs the `chctoboogie` plugin which does the translation from CHCs into a recursive program. We use the Boogie language to represent that program. Afterwards the default toolchain for verifying a recursive Boogie programs by ULTIMATE AUTOMIZER is applied. The ULTIMATE framework shares the libraries for handling SMT formulas with the SMTInterpol SMT solver. While verifying a program, ULTIMATE AUTOMIZER needs SMT solvers for checking satisfiability, for computing Craig interpolants and for computing unsatisfiable cores. The version of UNIHORN that participated in the competition used the SMT solvers SMTInterpol[8]and Z3[9]. The ULTIMATE framework is written in Java and build upon the Eclipse Rich Client Platform (RCP). The source code is available at GitHub[10].

Configuration in CHC-COMP-20. Our StarExec archive for the competition is shipped with the `bin/starexec_run_default` shell script calls the ULTIMATE command line interface with the `AutomizerCHC.xml` toolchain file and the `AutomizerCHC_No_Goto.epf` settings file. Both files can be found in toolchain (resp. settings) folder of ULTIMATE's repository.

`https://ultimate.informatik.uni-freiburg.de/`
LGPLv3 with a linking exception for Eclipse RCP

[8] `https://ultimate.informatik.uni-freiburg.de/smtinterpol/`
[9] `https://github.com/Z3Prover/z3`
[10] `https://github.com/ultimate-pa/`

Eldarica v2.0.3 (Hors Concours)

Hossein Hojjat
University of Tehran, Iran

Philipp Rümmer
Uppsala University, Sweden

Algorithm. Eldarica [15] is a Horn solver applying classical algorithms from model checking: predicate abstraction and counterexample-guided abstraction refinement (CEGAR). Eldarica can solve Horn clauses over linear integer arithmetic, arrays, algebraic data-types, and bit-vectors. It can process Horn clauses and programs in a variety of formats, implements sophisticated algorithms to solve tricky systems of clauses without diverging, and offers an elegant API for programmatic use.

Architecture and Implementation. Eldarica is entirely implemented in Scala, and only depends on Java or Scala libraries, which implies that Eldarica can be used on any platform with a JVM. For computing abstractions of systems of Horn clauses and inferring new predicates, Eldarica invokes the SMT solver Princess [22] as a library.

Configuration in CHC-COMP-20. Eldarica is in the competition run with the option `-abstractPO`, which enables a simple portfolio mode: two instances of the solver are run in parallel, one with the default options, and one with the option `-abstract:off` to switch off the interpolation abstract technique.

`https://github.com/uuverifiers/eldarica`
BSD licence

References

[1] Armin Biere, Alessandro Cimatti, Edmund Clarke & Yunshan Zhu (1999): *Symbolic Model Checking without BDDs*. In W. Rance Cleaveland, editor: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 193–207.

[2] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan & Andrey Rybalchenko (2015): *Horn Clause Solvers for Program Verification*. In: *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, pp. 24–51.

[3] Martin Blicha, Antti E. J. Hyvärinen, Jan Kofroň & Natasha Sharygina (2019): *Decomposing Farkas Interpolants*. In Tomáš Vojnar & Lijun Zhang, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer International Publishing, Cham, pp. 3–20.

[4] Martin Blicha, Antti E. J. Hyvärinen, Matteo Marescotti & Natasha Sharygina (2020): *A Cooperative Parallelization Approach for Property-Directed k-Induction*. In Dirk Beyer & Damien Zufferey, editors: *Verification, Model Checking, and Abstract Interpretation*, Springer International Publishing, Cham, pp. 270–292.

[5] Aaron R. Bradley (2011): *SAT-Based Model Checking without Unrolling*. In: *VMCAI*, *LNCS* 6538, Springer, pp. 70–87.

[6] Alessandro Cimatti, Alberto Griggio, Sergio Mover & Stefano Tonetta (2016): *Infinite-state invariant checking with IC3 and predicate abstraction*. *Formal Methods Syst. Des.* 49(3), pp. 190–218.

[7] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma & Roberto Sebastiani (2013): *The MathSAT5 SMT Solver*. In Nir Piterman & Scott Smolka, editors: *Proceedings of TACAS*, *LNCS* 7795, Springer.

[8] Daniel Dietsch, Matthias Heizmann, Jochen Hoenicke, Alexander Nutz & Andreas Podelski (2019): *Ultimate TreeAutomizer (CHC-COMP Tool Description)*. In: *HCVS/PERR@ETAPS*, *EPTCS* 296, pp. 42–47.

[9] Bruno Dutertre (2014): *Yices 2.2*. In Armin Biere & Roderick Bloem, editors: *Computer-Aided Verification (CAV'2014)*, *LNCS* 8559, Springer, pp. 737–744.

[10] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea & Andrey Rybalchenko (2012): *Synthesizing Software Verifiers from Proof Rules*. In: *PLDI*, ACM, pp. 405–416.

[11] Alberto Griggio (Accessed 2020): *Open-source IC3 Modulo Theories with Implicit Predicate Abstraction*. Available at `https://es-static.fbk.eu/people/griggio/ic3ia/index.html`.

[12] Matthias Heizmann, Yu-Fang Chen, Daniel Dietsch, Marius Greitschus, Jochen Hoenicke, Yong Li, Alexander Nutz, Betim Musa, Christian Schilling, Tanja Schindler & Andreas Podelski (2018): *Ultimate Automizer and the Search for Perfect Interpolants - (Competition Contribution)*. In: *TACAS (2)*, *LNCS* 10806, Springer, pp. 447–451.

[13] Matthias Heizmann, Jochen Hoenicke & Andreas Podelski (2013): *Software Model Checking for People Who Love Automata*. In: *CAV*, *LNCS* 8044, Springer, pp. 36–52.

[14] Jochen Hoenicke & Tanja Schindler (2018): *Efficient Interpolation for the Theory of Arrays*. In: *IJCAR*, *LNCS* 10900, Springer, pp. 549–565.

[15] Hossein Hojjat & Philipp Rümmer (2018): *The ELDARICA Horn Solver*. In Nikolaj Bjørner & Arie Gurfinkel, editors: *2018 Formal Methods in Computer Aided Design, FMCAD*, IEEE, pp. 1–7. Available at `https://ieeexplore.ieee.org/xpl/conhome/8585253/proceeding`.

[16] Antti E. J. Hyvärinen, Matteo Marescotti, Leonardo Alt & Natasha Sharygina (2016): *OpenSMT2: An SMT Solver for Multi-core and Cloud Computing*. In Nadia Creignou & Daniel Le Berre, editors: *Theory and Applications of Satisfiability Testing – SAT 2016*, Springer International Publishing, Cham, pp. 547–553.

[17] Dejan Jovanović & Bruno Dutertre (2016): *Property-directed k-induction*. In: *2016 Formal Methods in Computer-Aided Design (FMCAD)*, pp. 85–92.

[18] Hari Govind V K, YuTing Chen, Sharon Shoham & Arie Gurfinkel (forthcoming): *Global Guidance for Local Generalization in Model Checking*. In: *Computer Aided Verification - 32nd International Conference, CAV 2020.*

[19] Anvesh Komuravelli, Arie Gurfinkel & Sagar Chaki (2016): *SMT-based model checking for recursive programs*. *Formal Methods Syst. Des.* 48(3), pp. 175–205, doi:10.1007/s10703-016-0249-4. Available at `https://doi.org/10.1007/s10703-016-0249-4`.

[20] Daniel Kroening, Alex Groce & Edmund M. Clarke (2004): *Counterexample Guided Abstraction Refinement Via Program Execution*. In: *ICFEM*, *LNCS* 3308, Springer, pp. 224–238.

[21] Matteo Marescotti, Antti E. J. Hyvärinen & Natasha Sharygina (2018): *SMTS: Distributed, Visualized Constraint Solving*. In: *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018.*

[22] Philipp Rümmer (2008): *A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic*. In: *Proceedings, 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, *LNCS* 5330, Springer, pp. 274–289.

[23] Yuki Satake, Hiroshi Unno & Hinata Yanagi (2020): *Probabilistic Inference for Predicate Constraint Satisfaction*. In: *Proceedings of AAAI 2020.*

[24] Aaron Stump, Geoff Sutcliffe & Cesare Tinelli (2014): *StarExec: A Cross-Community Infrastructure for Logic Solving*. In Stéphane Demri, Deepak Kapur & Christoph Weidenbach, editors: *Automated Reasoning - 7th International Joint Conference, IJCAR*, *LNCS* 8562, Springer, pp. 367–373, doi:10.1007/978-3-319-08587-6_28. Available at `https://doi.org/10.1007/978-3-319-08587-6_28`.

[25] Xiaozhen Zhang & Weiqiang Kong (2020): *Facilitating CHC Solving with Improving Interpolation Abstraction Exploration*. To appear.

Solver	Score	#sat	#unsat	CPU time (s)	Wall-clock (s)	Speedup	SotAC
Spacer	554	292	262	6.03	6.11	0.99	0.28
Eldarica (HC)	513	265	248	43.58	19.10	2.28	0.23
Eldarica-abs	513	266	247	52.07	35.96	1.45	0.23
U. Unihorn Automizer	420	212	208	75.73	49.11	1.54	0.21
PCSat	331	156	175	92.10	29.54	3.12	0.20
U. TreeAutomizer	118	34	84	41.17	30.00	1.37	0.17
Any solver	560	298	262				

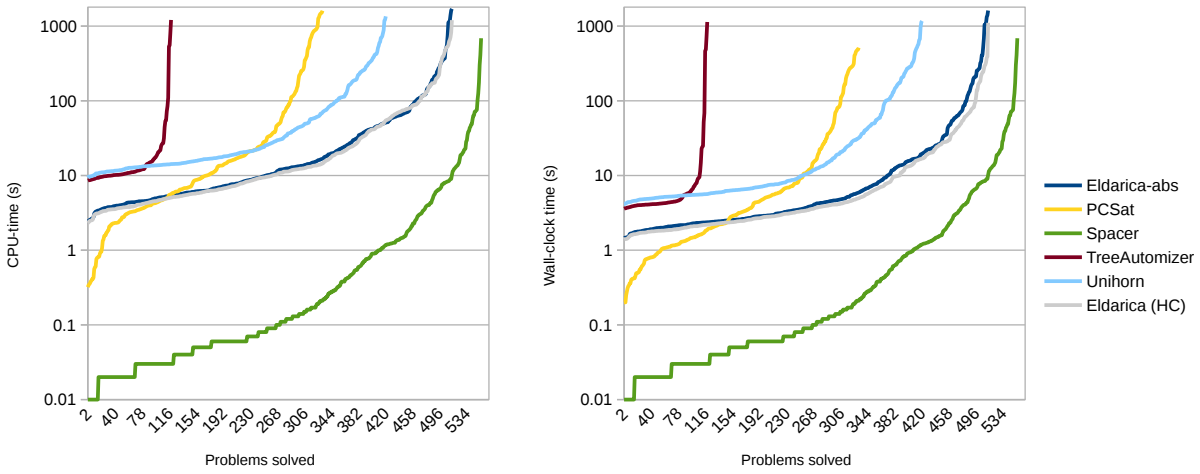

Figure 1: Solver performance on the 565 benchmarks of the LIA-nonlin track

Solver	Score	#sat	#unsat	CPU time (s)	Wall-clock (s)	Speedup	SotAC
Spacer	518	330	188	11.94	12.03	0.99	0.22
Eldarica-abs	477	300	177	57.26	39.59	1.45	0.20
Eldarica (HC)	476	300	176	48.58	20.00	2.43	0.20
U. Unihorn Automizer	407	240	167	43.57	26.21	1.66	0.17
IC3IA	400	260	140	46.09	46.23	1.00	0.20
PCSat	329	191	138	37.91	12.23	3.10	0.17
U. TreeAutomizer	307	166	141	50.30	37.43	1.34	0.17
Any solver	558	356	202				

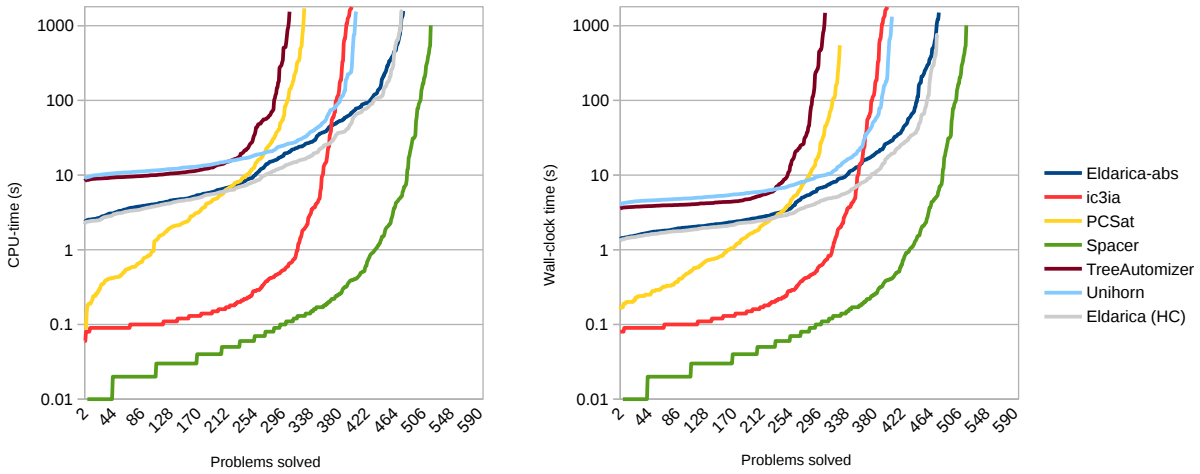

Figure 2: Solver performance on the 596 benchmarks of the LIA-lin track

Solver	Score	#sat	#unsat	CPU time (s)	Wall-clock (s)	Speedup	SotAC
Spacer	295	203	92	0.81	0.89	0.91	0.37
U. Unihorn Automizer	217	144	73	39.73	24.12	1.65	0.26
ProphIC3	214	140	74	38.24	19.17	1.99	0.34
IC3IA	147	92	55	9.17	9.30	0.99	0.24
U. TreeAutomizer	147	100	47	31.49	21.46	1.47	0.22
Eldarica (HC)	91	91	0	106.80	68.05	1.57	0.24
Any solver	350	250	100				

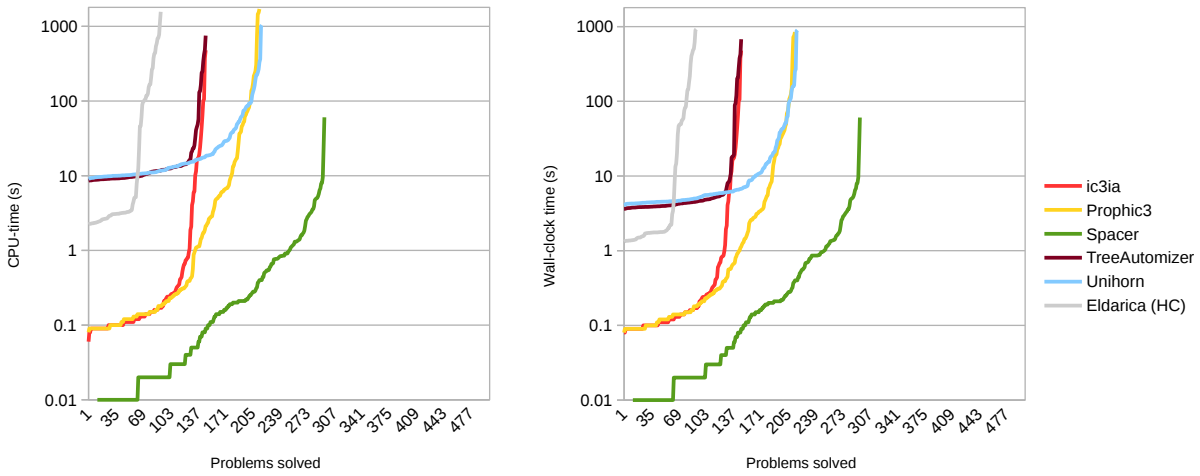

Figure 3: Solver performance on 500 benchmarks of the LIA-lin-arrays track (one benchmark on which Spacer and Ultimate Unihorn Automizer give conflicting answers is not counted)

Solver	Score	#sat	#unsat	CPU time (s)	Wall-clock (s)	Speedup	SotAC
IC3IA	468	378	90	136.94	137.05	1.00	0.29
Sally-parallel	439	360	79	138.81	47.37	2.93	0.24
Sally-decomposing-itp	438	357	81	107.61	107.68	1.00	0.24
Spacer	346	270	76	176.75	176.86	1.00	0.22
U. TreeAutomizer	168	131	37	239.75	202.11	1.19	0.19
U. Unihorn Automizer	160	103	57	213.33	158.57	1.35	0.18
Any solver	481	388	93				

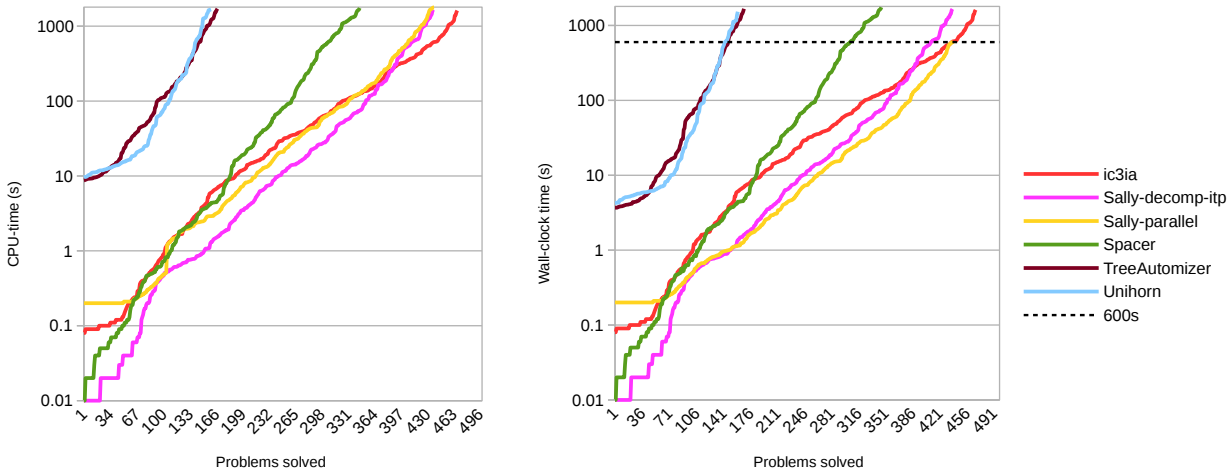

Figure 4: Solver performance on the 499 benchmarks of the LRA-TS track