Competition Report: CHC-COMP-21

Grigory Fedyukovich
Florida State University, USA

Philipp Rümmer
Uppsala University, Sweden

CHC-COMP-21[1] is the fourth competition of solvers for Constrained Horn Clauses. In this year, 7 solvers participated at the competition, and were evaluated in 7 separate tracks on problems in linear integer arithmetic, linear real arithmetic, arrays, and algebraic data-types. The competition was run in March 2021 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 [3, 1]. 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 techniques including model checking, abstract interpretation, or clause transformation. 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 reactive systems) 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 verification tools, and thus enable construction of verification systems in a modular way.

CHC-COMP-21 is the fourth competition of solvers for Constrained Horn Clauses, a competition affiliated with the 8th Workshop on Horn Clauses for Verification and Synthesis (HCVS) at ETAPS 2021. 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-21 was March 18 2021, resulting in 7 solvers participating, which were evaluated in the second half of March 2021. The 7 solvers were evaluated in 7 separate tracks on problems in linear integer arithmetic, linear real arithmetic, the theory of arrays, and theories of algebraic data-types. The results of the competition can be found in Section 6 of this report, and were presented at the (virtual) HCVS workshop on March 28 2021.

1.1 Acknowledgements

We would like to thank the HCVS chairs, Bishoksan Kafle and Hossein Hojjat, for hosting CHC-COMP also in this year!

CHC-COMP-21 heavily built on the infrastructure developed for the previous instances of CHC-COMP, run by Arie Gurfinkel, Grigory Fedyukovich, and Philipp Rümmer, respectively. Contributors to the competition infrastructure also include Adrien Champion, Dejan Jovanovic, and Nikolaj Bjørner.

Like in the first three competitions, CHC-COMP-21 was run on StarExec [20]. 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 2021

and competent support by Aaron Stump and his team whenever problems occurred. CHC-COMP-21 would not have been possible without this!

Philipp Rümmer is supported by the Swedish Research Council (VR) under grant 2018-04727, by the Swedish Foundation for Strategic Research (SSF) under the project WebSec (Ref. RIT17-0011), and by the Knut and Alice Wallenberg Foundation under the project UPDATE.

2 Brief Overview of the Competition Design

2.1 Competition Tracks

Three new tracks were introduced in CHC-COMP-21 (namely, LIA-nonlin-arrays, LRA-TS-par, ADT-nonlin), leading to altogether 7 tracks:

- **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-nonlin-arrays**: benchmarks with at least one non-linear clause, and the combined theory of linear integer arithmetic and arrays 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;
- **LRA-TS-par**: same selection of benchmarks as in LRA-TS, but 2x4 CPU cores were reserved for each task, and the evaluation was done with wall-clock time limit; this yields a setting benefiting parallel solvers;
- **ADT-nonlin**: benchmarks with at least one non-linear clause, and the algebraic data-types as background theory.

2.2 Computing Nodes

Two separate queues on StarExec were used for the competition, one queue with 15 nodes for the track LRA-TS-par, and one with 20 nodes for all other tracks. Each node had two quadcore CPUs. In LRA-TS-par, each job was run on its own node during the competition runs, while in the other tracks each node was used to run two jobs in parallel. The machine specifications 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

The solvers submitted to CHC-COMP-21 were evaluated twice:

- in a first set of **test runs**, in which (optional) 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. 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-21. 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 run of LRA-TS-par, each job was limited to 1800s wall-clock time, and 64GB memory. In the competition runs of all other tracks, each job was limited to 1800s CPU time, 1800s wall-clock time, and 64GB memory.

2.4 Evaluation of the Competition Runs

The evaluation of the competition runs was in this year done using the `summarize.py` script available in the repository `https://github.com/chc-comp/scripts`, and on the basis of the data provided by StarExec through the "job information" data export function. 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** (for LRA-TS-par, by **Wall-clock 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**, **CPU time**, and **Wall-clock time** are:

- **Score**: the number of **sat** or **unsat** results produced by a solver on the benchmarks of a track.
- **CPU time**: the total CPU time needed by a solver to produce its answers in some track, including **unknown** answers.
- **Wall-clock time**: the total wall-clock time needed by a solver to produce its answers in some track, including **unknown** answers.

In addition, the following feature is included in the results for each solver and each track:

- **#unique**: The number of **sat** or **unsat** results produced by a solver for benchmarks for which all other solvers returned **unknown**.

We decided to not include the **Space** feature, specifying the total maximum virtual memory consumption, in the tables, since this number is less telling for solvers running in a JVM.

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`. 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 in Tracks other than ADT-nonlin

All benchmarks used in CHC-COMP-21 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` has the effect of merging multiple queries in a benchmark into a single query by introducing an auxiliary nullary predicate. This transformation was introduced in CHC-COMP-20, and is discussed in [18].

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`. Benchmarks that did not conform to any of the competition tracks, were not used in CHC-COMP-21.

3.3 Benchmark Processing in ADT-nonlin

Benchmarks used in the ADT-nonlin track were preprocessed by eliminating all theory constraints and recursively-defined functions. The transformation was performed using the feature of the RInGen tool [13]. This way, we were able to satisfy the input-language constraints for all four tools entering the competition in this track. In the future, we, however, plan introducing other ADT-related tracks with benchmarks over ADT and linear arithmetic and/or arrays.

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 and Section 3.3. 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.

For ADT-nonlin, benchmarks originate from the TIP suite (originally, designed for theorem-proving) and verification of programs in functional languages.

4 Benchmark Rating and Selection

This section describes how the benchmarks for CHC-COMP-21 were selected among the unique benchmarks summarised in Table 1. For the competition tracks LIA-lin-arrays, LRA-TS, and ADT-nonlin, the benchmark library only contains 488, 498, and 506 unique benchmarks, respectively, which are small

Repository	LIA-nonlin	LIA-lin	LIA-nonlin-arrays	LIA-lin-arrays	LRA-TS	ADT-nonlin
adt-purified						67 / 67
aeval		54 / 54				
eldarica-misc	69 / 66	147 / 134				
extra-small-lia		55 / 55				
hcai	135 / 133	100 / 86	25 / 25	39 / 39		
hopv	68 / 67	49 / 48				
jayhorn	5138 / 5084	75 / 73				
kind2	851 / 738					
ldv-ant-med			79 / 79	10 / 10		
ldv-arrays			821 / 546	3 / 2		
llreve	59 / 57	44 / 44		31 / 31		
quic3				43 / 43		
ringen						439 / 439
sally					177 / 174	
seahorn	68 / 66	3396 / 2822				
synth/nay-horn	119 / 114					
synth/semgus			5371* / 4839*			
tricera	4 / 4	405 / 405				
vmt		905 / 802			99 / 98	
chc-comp19	271 / 265	325 / 313	15 / 15	290 / 290	228 / 226	
sv-comp	1643 / 1169	3150 / 2932	855 / 779	79 / 73		
Total	8425 / 7763	8705 / 7768	7166 / 6283	495 / 488	504 / 498	506 / 506

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-21 track, the first number gives the total number of benchmarks, and the second number the number of contributed unique benchmarks (after discarding duplicate benchmarks). In the benchmark family synth/semgus, only 2357/2331 benchmarks were taken into account for the competition, as processing of the other benchmarks (according to Section 3.2) was incorrect due to inconsistent filename conventions. This mistake was only discovered after the main competition runs were finished, and could not be corrected in time.

enough sets to use all benchmarks in the competition. For the tracks LIA-nonlin, LIA-lin, and LIA-nonlin-arrays, 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, LIA-lin, LIA-nonlin-arrays, a simple rating based on the performance of the CHC-COMP-20 solvers was computed. The same approach was used in the last competition, CHC-COMP-20, using solvers from CHC-COMP-19. In this year, the two top-ranked competing solvers from CHC-COMP-20 were run for a few seconds on each of the benchmarks:[2]

- For **LIA-nonlin** and **LIA-lin**: Spacer (timeout 5s) and Eldarica-abs (timeout 10s);
- For **LIA-nonlin-arrays**: Spacer (timeout 5s) and Ultimate Unihorn (timeout 10s). Since LIA-nonlin-arrays was not evaluated at CHC-COMP-20, the top-ranked solvers from the track LIA-lin-arrays were chosen.

All solvers were run using the same binary and same options as in CHC-COMP-20. For the JVM-based tools, Eldarica-abs and Ultimate Unihorn, the higher timeout was chosen to compensate for the JVM

[2]Run on an Intel Core i5-650 2-core machine with 3.2GHz. All timeouts are in terms of wall-clock time.

	LIA-nonlin			LIA-lin			LIA-nl-arrays		
Repository	#A /	#B /	#C	#A /	#B /	#C	#A /	#B /	#C
aeval				11 /	15 /	28			
eldarica-misc	35 /	4 /	27	105 /	20 /	9			
extra-small-lia				21 /	24 /	10			
hcai	74 /	44 /	15	73 /	8 /	5	14 /	6 /	5
hopv	60 /	7 /		47 /	1 /				
jayhorn	2688 /	769 /	1627	73 /	/				
kind2	250 /	455 /	33						
ldv-ant-med							/	25 /	54
ldv-arrays							/	127 /	419
llreve	35 /	13 /	9	37 /	5 /	2			
seahorn	38 /	19 /	9	977 /	985 /	860			
synth/nay-horn	46 /	30 /	38						
synth/semgus							282 /	768 /	1281
tricera	4 /	/		28 /	14 /	363			
vmt				85 /	616 /	101			
chc-comp19	144 /	80 /	41	80 /	101 /	132	/	7 /	8
sv-comp	1013 /	144 /	12	2801 /	17 /	114	258 /	268 /	253
Total	4387 /	1565 /	1811	4338 /	1806 /	1624	554 /	1201 /	2020

Table 2: The number of unique benchmarks with ratings A / B / C, respectively.

start-up delay.

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.

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-21 can be found in the github repository `https://github.com/chc-comp/chc-comp21-benchmarks`, and on StarExec in the public space `CHC/CHC-COMP/chc-comp21-benchmarks`.

	LIA-nonlin		LIA-lin		LIA-nl-arrays		LIA-lin-arrays	LRA-TS	ADT-nonlin
Repository	N_r	#Sel	N_r	#Sel	N_r	#Sel	#Selected	#Selected	#Selected
adt-purified									67
aeval			10	30					
eldarica-misc	10	30	15	39					
extra-small-lia			10	30					
hcai	20	55	15	28	5	15	39		
hopv	10	17	10	11					
jayhorn	30	90	10	10					
kind2	30	90							
ldv-ant-med					20	60	10		
ldv-arrays					30	90	2		
llreve	15	37	10	17			31		
quic3							43		
ringen									111
sally								174	
seahorn	15	39	30	90					
synth/nay-horn	20	60							
synth/semgus	20	60			45	135			
tricera	1	1	20	60					
vmt			30	90				98	
chc-comp19	30	90	30	90	5	15	290	226	
sv-comp	30	72	30	90	45	135	73		
Total		581		585		450	488	498	178

Table 3: The number of selected unique benchmarks for the CHC-COMP-21 tracks.

5 Solvers Entering CHC-COMP-21

In total, 7 solvers were submitted to CHC-COMP-21: 6 competing solvers, and one further solver (Eldarica, co-developed by one of the competition organisers) 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-comp21-benchmarks`.

6 Competition Results

The winners and top-ranked solvers of the seven CHC-COMP-21 tracks are:

	LIA-nonlin	LIA-lin	LIA-nl-arrays	LIA-lin-arrays	LRA-TS	LRA-TS-par	ADT-nonlin
Winner	**Spacer**	**Spacer**	**Spacer**	**Spacer**	**Spacer**	**Spacer**	**Spacer**
Place 2	Ultimate Unihorn	Golem	Ultimate Unihorn	Ultimate Unihorn	Golem	Golem	RInGen
Place 3	PCSat	Ultimate Unihorn	Ultimate TreeAutomizer	Ultimate TreeAutomizer	Ultimate TreeAutomizer	Ultimate TreeAutomizer	PCSat

Detailed results for the seven tracks are provided in the tables on page 18.

Solver	LIA-nonlin	LIA-lin	LIA-nonlin-arrays	LIA-lin-arrays	LRA-TS	LRA-TS-par	ADT-nonlin
Golem	—	`LIA-Lin`	—	—	`LRA-TS`	`LRA-TS`	—
PCSat	`pcsat_-tb_-ucore_ar`	`pcsat_-tb_-ucore_ar`	—	—	—	—	`pcsat_-tb_-ucore_-reduce_-quals`
Spacer	`LIA-NONLIN`	`LIA-LIN`	`LIA-NONLIN-ARRAYS`	`LIA-LIN-ARRAYS`	`LRA-TS`	`LRA-TS`	`ADT-NONLIN`
Ultimate TreeAutomizer	`default`	`default`	`default`	`default`	`default`	`default`	—
Ultimate Unihorn	`default`	`default`	`default`	`default`	`default`	`default`	—
RInGen	—	—	—	—	—	—	`default`
Eldarica (Hors Concours)	`def`	`def`	`def`	`def`	—	—	`def`

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

6.1 Observed Issues and Fixes during the Competition Runs

Fixes in Spacer: During the competition runs, it was observed that Spacer, in the version submitted by March 18, did not run correctly on StarExec and did not produce output for any of the benchmarks. Since this issue was discovered soon after the start of the competition runs, the organisers decided to let the Spacer authors submit a corrected version. The problem turned out to be compilation/linking-related, and the results presented in this report were produced with the fixed version of Spacer. To ensure fairness of the competition, all teams were given time until March 20 to submit revised versions of their tools.

Fixes in Golem: One case of inconsistent results was observed in the competition runs in the track LIA-lin. For the benchmark `chc-LIA-Lin_502.smt2`, the tool Golem reported **unsat**, while Spacer and Eldarica reported **sat**. The author of Golem confirmed that the inconsistency was due to a bug in the loop acceleration in Golem, and could provide a corrected version in which loop acceleration was switched off. The results presented in this report were produced with this fixed version of Golem.

To ensure fairness of the competition, we provide the following table comparing the results of the two versions of Golem in track LIA-lin. The table shows that the fix in Golem led to marginally worse performance of the solver, and therefore did not put the other solvers at an unfair disadvantage:

	#sat	#unsat
Golem (original)	185	133
Golem (fixed)	179	133

7 Conclusions

The organisers would like to congratulate the general winner of this year's CHC-COMP, the solver **Spacer**, as well as all solvers and tool authors for their excellent performance! Thanks go to everybody who has 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!

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

- **Models and counterexamples** (as already discussed in [18]). A concern brought up again at the HCVS workshop is the generation of models and/or counterexample certificates, highlighting the user demand for this functionality. Since at the moment many tools do not support certificates yet, this could initially happen in the scope of a new track, or by awarding a higher number of points for each produced and verified model/counterexample.
- **Multi-query benchmarks** (as already discussed in [18]). We propose to extend the CHC-COMP fragment of SMT-LIB to include also problems with multiple queries. This would leave the decision how to handle multi-query benchmarks to each solver. For solvers that can only solve problems with a single query, a script is available to transform multi-query problems to single-query problems.
- **The LRA-TS track.** This restricted track was created to enable also solvers that only support traditional transition systems to enter. However, no such solver was submitted to CHC-COMP-21 (in contrast to CHC-COMP-20), which means that the results presented in this report do not fully reflect the state of the art for such problems. For future instances of CHC-COMP, it can be considered to replace LRA-TS with a general LRA track, dropping the restriction to problems in transition system form.
- **ADT-nonlin**: As mentioned in Sect. 3.3, the syntactic restrictions on the ADT tasks were needed to let more solvers participate in the competition. Thus, we had only "pure ADT" problems. However, as the technology evolves, we expect more solvers to participate in the next editions of the competition. Thus, ADT tasks that also use constraints in other theories (if collected in sufficient amounts) could form new tracks.
- **A bigger set of benchmarks is needed, and all users and tool authors are encouraged to submit benchmarks!** In particular, in the LIA-nonlin, LRA-TS, and ADT-nonlin tracks, the competition results indicate that more and/or 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.

GOLEM

Martin Blicha
Università della Svizzera italiana, Switzerland

Algorithm. GOLEM is a new CHC solver, still under active development. It can solve systems of linear clauses with Linear Real or Integer Arithmetic as the background theory and it is able to provide witnesses for both satisfiable and unsatisfiable systems.

Its current reasoning engine is a re-implementation of the IMPACT algorithm [15] and thus falls into the category of interpolation-based model-checking approaches.

Architecture and Implementation. GOLEM is implemented in C++ and built on top of the interpolating SMT solver OPENSMT [9] which is used for both satisfiability solving and interpolation. The only dependencies are those inherited from OPENSMT: Flex, Bison and GMP libraries.

Configuration in CHC-COMP-21. GOLEM was run with its default settings, except that its experimental loop acceleration module had to be disabled, because it contained a bug in the submitted version. Note that the SMT theory needs to be specified.

```
$ golem --logic QF_LRA --accelerate-loops=false
$ golem --logic QF_LIA --accelerate-loops=false
```

`http://verify.inf.usi.ch/golem`
MIT LICENSE

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. Its applications include but not limited to branching-time temporal verification, relational 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 [21], decision-tree-based [14], and graphical-model-based [19] ones.

Architecture and Implementation. PCSat is designed and implemented as a highly-configurable solver, allowing us to test various combinations of synthesis engines, example sampling methods, template refinement strategies, and qualifier generators. This design is enabled by a powerful module system and metaprogramming features of the OCaml functional programming language. PCSat uses Z3 as the backend SMT solver.

News in 2021. We supported the theory of algebraic datatypes and implemented a preprocessor for eliminating irrelevant arguments of predicates.

Configuration in CHC-COMP-21. PCSat is run with the solver configuration file "pcsat_tb_ucore_-ar.json" in the LIA-Nonlin and LIA-Lin tracks and "pcsat_tb_ucore_reduce_quals.json" in the ADT-Nonlin track. Both configurations enable the template-based synthesis engine.

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

RInGen v1.1

Yurii Kostyukov
Saint Petersburg State University, JetBrains Research, Russia
Dmitry Mordvinov
Saint Petersburg State University, JetBrains Research, Russia

Algorithm. RInGen stands for a *R*egular *In*variant *Gen*erator, where *regular invariants* [13] are represented by *finite tree automata*. While invariant representations based on first-order logic (FOL) can only access finitely many subterms, regular invariants have an ability to "scan" an ADT term to the unbounded depth via automaton rules. Tree automata also enjoy useful decidability properties and the corresponding regular tree languages are closed under all set operations, which makes regular invariants a promising alternative to FOL-based invariant representations.

RInGen rewrites a system of CHCs over ADTs into a formula over uninterpreted function symbols by eliminating all disequalities, testers, and selectors from the clause bodies. Then the satisfiability modulo theory of ADTs is reduced to satisfiability modulo theory of uninterpreted functions with equality (EUF). After that, an off-the-shelf finite model finder is applied to build a finite model of the reduced verification conditions. Finally, using the correspondence between finite models and tree automata, the automaton representing the safe inductive invariant of the original system is obtained. Full algorithmic details of the RInGen can be found in [13].

Architecture and Implementation. RInGen accepts input in the SMTLIB2 format and produces CHCs over pure ADT sorts in SMTLIB2 and Prolog. It takes conditions with a property as input and checks if the property holds, returning SAT and a safe inductive invariant, or terminates with UNSAT otherwise. We exploit CVC4 (using `cvc4 --finite-model-find`) at the backend to find regular models. Besides regular models, a finite model finding approach of CVC4 [16] v1.8 based on quantifier instantiation provides us with sound satisfiability checking.

Configuration in CHC-COMP-21. The tool is built and run with the following arguments:

```
solve --timelimit $tlimit --quiet --output-directory "$dir" cvc4f "$input".
```

`https://github.com/Columpio/RInGen/releases/tag/v1.1`
BSD 3-Clause License

SPACER

Hari Govind V K
University of Waterloo, Canada

Arie Gurfinkel
University of Waterloo, Canada

Algorithm. SPACER [12] 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 POBs 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 [10, 11].

The current implementation of SPACER supports linear and non-linear CHCs in the theory of Arrays, Linear Arithmetic, FixedSizeBitVectors, and Algebraic Data Types. 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 an interpolating SMT solver.

Configuration in CHC-COMP-21. SPACER has several configurations. The following options are common to all configurations:

```
fp.xform.tail_simplifier_pve=false fp.validate=true
fp.spacer.mbqi=false fp.spacer.use_iuc=true
```

To activate global guidance [10], we use the following options:

```
fp.spacer.global=true fp.spacer.concretize=true
fp.spacer.conjecture=true fp.spacer.expand_bnd=true
```

To activate quantifier generalization [4], we use:

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

In the arithmetic tracks (LRA-TS, LIA-LIN, LIA-NONLIN), we ran two threads in parallel. The first thread ran SPACER with global guidance. The second thread ran Z3's BMC engine:

```
fp.engine=bmc
```

In the array tracks (LIA-LIN-ARRAYS, LIA-NONLIN-ARRAYS), we again ran two threads in parallel. The first thread had both global guidance and quantifier generalization. The second thread had only quantifier generalization. In the ADT tracks (ADT-LIN, ADT-NONLIN), we ran one thread which used only global guidance. Additionally, for the ADT tracks, we turned off one of the optimizations in SPACER:

```
fp.spacer.use_inc_clause=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 [2]. 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[3] [7]. For checking satisfiability, we use the Z3 SMT solver[4]. The tree automata are implemented in ULTIMATE's automata library[5]. The ULTIMATE framework is written in Java and build upon the Eclipse Rich Client Platform (RCP). The source code is available at GitHub[6].

Configuration in CHC-COMP-21. 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

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

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 [5] which implements an automata-based approach to software verification [6].

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[7]and Z3[8]. The ULTIMATE framework is written in Java and build upon the Eclipse Rich Client Platform (RCP). The source code is available at GitHub[9].

Configuration in CHC-COMP-21. 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

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

Eldarica v2.0.6 (Hors Concours)

Zafer Esen
Uppsala University, Sweden

Hossein Hojjat
University of Tehran, Iran

Philipp Rümmer
Uppsala University, Sweden

Algorithm. Eldarica [8] 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 [17] as a library.

News in 2021. Compared to the last competition, Eldarica now uses a new array solver in the tracks LIA-nonlin-arrays and LIA-lin-arrays.

Configuration in CHC-COMP-21. 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 abstraction technique.

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

References

[1] 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, doi:10.1007/978-3-319-23534-9_2.

[2] 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, doi:10.4204/EPTCS.296.7.

[3] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea & Andrey Rybalchenko (2012): *Synthesizing Software Verifiers from Proof Rules*. In: *PLDI*, ACM, pp. 405–416, doi:10.1145/2254064.2254112.

[4] Arie Gurfinkel, Sharon Shoham & Yakir Vizel (2018): *Quantifiers on Demand*. In Shuvendu K. Lahiri & Chao Wang, editors: *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, *Lecture Notes in Computer Science* 11138, Springer, pp. 248–266, doi:10.1007/978-3-030-01090-4_15. Available at `https://doi.org/10.1007/978-3-030-01090-4_15`.

[5] 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, doi:10.1007/978-3-319-89963-3_30.

[6] Matthias Heizmann, Jochen Hoenicke & Andreas Podelski (2013): *Software Model Checking for People Who Love Automata*. In: *CAV*, *LNCS* 8044, Springer, pp. 36–52, doi:10.1007/978-3-642-39799-8_2.

[7] Jochen Hoenicke & Tanja Schindler (2018): *Efficient Interpolation for the Theory of Arrays*. In: *IJCAR*, *LNCS* 10900, Springer, pp. 549–565, doi:10.1007/978-3-319-94205-6_36.

[8] 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, doi:10.23919/FMCAD.2018.8603013.

[9] 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.

[10] 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*.

[11] Hari Govind V. K., Grigory Fedyukovich & Arie Gurfinkel (2020): *Word Level Property Directed Reachability*. In: *IEEE/ACM International Conference On Computer Aided Design, ICCAD 2020, San Diego, CA, USA, November 2-5, 2020*, IEEE, pp. 107:1–107:9, doi:10.1145/3400302.3415708. Available at `https://doi.org/10.1145/3400302.3415708`.

[12] 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`.

[13] Yurii Kostyukov, Dmitry Mordvinov & Grigory Fedyukovich (2021): *Beyond the Elementary Representations of Program Invariants over Algebraic Data Types*. In: *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, pp. 451–465.

[14] Satoshi Kura, Hiroshi Unno & Ichiro Hasuo (2021): *Decision Tree Learning in CEGIS-Based Termination Analysis*. In: *Proceedings of CAV 2021*, Springer. To appear.

[15] Kenneth L. McMillan (2006): *Lazy Abstraction with Interpolants*. In Thomas Ball & Robert B. Jones, editors: *Computer Aided Verification*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 123–136.

[16] Andrew Reynolds, Cesare Tinelli, Amit Goel & Sava Krstić (2013): *Finite model finding in SMT*. In: *International Conference on Computer Aided Verification*, *Lecture Notes in Computer Science* 8044, Springer, pp. 640–655.

[17] 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, doi:10.1007/978-3-540-89439-1_20.

[18] Philipp Rümmer (2020): *Competition Report: CHC-COMP-20*. In Laurent Fribourg & Matthias Heizmann, editors: *Proceedings 8th International Workshop on Verification and Program Transformation and 7th Workshop on Horn Clauses for Verification and Synthesis, VPT/HCVS@ETAPS 2020, Dublin, Ireland, 25-26th April 2020*, *EPTCS* 320, pp. 197–219, doi:10.4204/EPTCS.320.15. Available at `https://doi.org/10.4204/EPTCS.320.15`.

[19] Yuki Satake, Hiroshi Unno & Hinata Yanagi (2020): *Probabilistic Inference for Predicate Constraint Satisfaction*. In: *Proceedings of AAAI 2020*, AAAI Press, pp. 1644–1651.

[20] 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.

[21] Hiroshi Unno, Tachio Terauchi & Eric Koskinen (2021): *Constraint-based Relational Verification*. In: *Proceedings of CAV 2021*, Springer. To appear.

Table 5: Solver performance on the 581 benchmarks of the LIA-nonlin track

Solver	Score	#sat	#unsat	CPU time/s	Wall-clock/s	#unique
Spacer	550	352	198	70325	35613	80
Eldarica	461	285	176	252300	120997	5
U. Unihorn	295	175	120	562355	471683	0
PCSat	280	172	108	555680	553500	0
U. TreeAutomizer	56	21	35	463165	433393	0

Table 6: Solver performance on the 585 benchmarks of the LIA-lin track

Solver	Score	#sat	#unsat	CPU time/s	Wall-clock/s	#unique
Spacer	484	310	174	196494	98301	41
Eldarica	397	258	139	367839	200046	27
Golem	312	179	133	475962	481432	2
U. Unihorn	301	175	126	498519	410585	0
PCSat	278	181	97	570522	538048	1
U. TreeAutomizer	207	106	101	580853	547293	0

Table 7: Solver performance on the 450 benchmarks of the LIA-nonlin-arrays track

Solver	Score	#sat	#unsat	CPU time/s	Wall-clock/s	#unique
Spacer	379	224	155	117861	62320	135
Eldarica	225	135	90	415182	200473	10
U. Unihorn	205	108	97	312415	250274	0
U. TreeAutomizer	88	21	67	299039	273337	1

Table 8: Solver performance on the 488 benchmarks of the LIA-lin-arrays track

Solver	Score	#sat	#unsat	CPU time/s	Wall-clock/s	#unique
Spacer	288	214	74	353882	177769	89
Eldarica	225	149	76	441240	208321	11
U. Unihorn	219	146	73	462492	402093	1
U. TreeAutomizer	147	100	47	387636	369112	0

Table 9: Solver performance on the 498 benchmarks of the LRA-TS track

Solver	Score	#sat	#unsat	CPU time/s	Wall-clock/s	#unique
Spacer	311	228	83	364422	188832	51
Golem	276	200	76	416173	416193	11
U. TreeAutomizer	169	132	37	632058	606468	15
U. Unihorn	160	103	57	646526	545329	1

Table 10: Solver performance on the 498 benchmarks of the LRA-TS-par track

Solver	Score	#sat	#unsat	CPU time/s	Wall-clock/s	#unique
Spacer	335	250	85	642476	329157	61
Golem	276	200	76	416152	416229	6
U. TreeAutomizer	169	132	37	652793	626474	14
U. Unihorn	166	109	57	804867	630927	1

Table 11: Solver performance on the 178 benchmarks of the ADT-nonlin track

Solver	Score	#sat	#unsat	CPU time/s	Wall-clock/s	#unique
Spacer	83	30	53	173478	173499	10
Eldarica	77	38	39	182413	87101	8
RInGen	71	26	45	194180	194199	6
PCSat	64	33	31	209696	209725	6