CHC-COMP 2023: Competition Report

Emanuele De Angelis*
IASI-CNR, Italy
emanuele.deangelis@iasi.cnr.it

Hari Govind V K
University of Waterloo, Canada
hgvk94@gmail.com

CHC-COMP 2023 is the sixth edition of the Competition of Solvers for Constrained Horn Clauses. The competition was run in April 2023 and the results were presented at the 10th Workshop on Horn Clauses for Verification and Synthesis held in Paris, France, on April 23, 2023. This edition featured seven solvers (six competing and one hors concours) and six tracks, each of which dealing with a class of clauses. This report describes the organization of CHC-COMP 2023 and presents its results.

1 Introduction

Constrained Horn Clauses (CHCs) are a class of first-order logic formulas where the Horn clause format is extended with *constraints*, that is, formulas of an arbitrary, possibly non-Horn, background theory (such as linear integer arithmetic, arrays, and algebraic data types).

CHCs have gained popularity as a formalism well suited for automatic program verification [20, 5, 9]). Indeed, the last decade has seen impressive progress in the development of solvers for CHCs (CHC solvers), which can now be effectively used as back-end tools for program verification due to their ability to solve satisfiability problems dealing with a variety of background theories. A non-exhaustive list of solvers includes: ADTInd [40], ADTRem [11], Eldarica [25], FreqHorn [16], Golem [7], HSF [20], PCSat [39], RAHFT [27], RInGen [29], SPACER [28], Ultimate TreeAutomizer [13], and VeriMAP [10].

CHC-COMP is an annual competition that aims to evaluate state-of-the-art CHC solvers on realistic and publicly available benchmarks; it is open to proposals and contributions from users and developers of CHC solvers, as well as researchers working in the field of CHC solving foundations and its applications.

CHC-COMP 2023[1] is the 6th edition of the CHC-COMP, affiliated with the 10th Workshop on Horn Clauses for Verification and Synthesis (HCVS 2023[2]) held in Paris, France, on April 23, 2023. The deadline for submitting candidate benchmarks was March 24, 2023. The deadlines for submitting tools for the test (optional) and the competition runs were 31 March and 7 April 2023, respectively. The competition was run in the subsequent two weeks, and the results were announced at HCVS 2023. CHC-COMP 2023 featured 7 solvers (6 competing solvers and one hors concours), and 6 tracks, each of which dealing with a class of clauses consisting of linear and nonlinear CHCs with constraints over linear integer arithmetic, arrays, non-recursive/recursive algebraic data types, and a few combinations thereof.

This report is structured as follows. Section 2 presents the competition tracks, the technical resources used to run the competition, and the evaluation model adopted to rank the solvers. Section 3 presents the inventory of benchmarks and how the candidate benchmarks have been processed and selected for the competition runs. Sections 4 and 5 present the tools submitted to CHC-COMP 2023 and the results of the competition, respectively. Section 6 presents some closing remarks from the organizers and participants of CHC-COMP 2023. Section 7 collects the tool descriptions contributed by the participants. Finally, Appendix A includes the tables with the detailed results about the competition runs.

*The author is member of the INdAM Research Group GNCS.
[1] https://chc-comp.github.io/
[2] https://www.sci.unich.it/hcvs23/

Submitted to:
HCVS 2023

Acknowledgements

We would also like to thank the HCVS 2023 Program Chairs, David Monniaux and Jose F. Morales, for hosting the competition this year as well, and all the HCVS attendees for the fruitful discussion we had after the presentation of the CHC-COMP report. A special thanks goes to Hossein Hojjat for presenting CHC-COMP 2023 at TOOLympics 2023[3].

CHC-COMP 2023 heavily built on the infrastructure developed by the organizers of the previous editions, that is, Grigory Fedyukovich, Arie Gurfinkel, and Philipp Rümmer, which also includes the contributions from Nikolaj Bjørner, Adrien Champion, and Dejan Jovanovic.

We are also extremely grateful to StarExec[4] [38] that continues to support the CHC community by providing the CHC-COMP the computing resources to run the competition. In particular, we would like to thank Aaron Stump for helping us in accessing and using the StarExec services.

2 Design and Organization

This section presents (i) the competition tracks, (ii) the technical resources used to run the solvers, (iii) the characteristics of the test and the competition runs, and (iv) the evaluation model used to rank the solvers in each track.

2.1 Tracks

CHC-COMP is organized in tracks, each of which deals with a class of CHCs. CHCs are classified according to the following features: (i) the background theory of the constraints, and (ii) the number of *uninterpreted atoms* (that is, atoms whose predicate symbols do not belong to the background theory) occurring in the premises of clauses. A clause with at most one uninterpreted atom in the premise is said to be *linear*, and *nonlinear* otherwise.

Solvers participating in the CHC-COMP 2023 could enter the competition in six tracks (one track was introduced in this edition, that is, ADT-LIA-nonlin, while the remaining tracks were inherited from the previous edition)[5]:

1. **LIA-lin**: Linear Integer Arithmetic - linear clauses
2. **LIA-nonlin**: Linear Integer Arithmetic - nonlinear clauses
3. **LIA-lin-Arrays**: Linear Integer Arithmetic & Arrays - linear clauses
4. **LIA-nonlin-Arrays**: Linear Integer Arithmetic & Arrays - nonlinear clauses
5. **LIA-nonlin-Arrays-nonrecADT**: Linear Integer Arithmetic & Arrays & nonrecursive Algebraic Data Types - nonlinear clauses
6. **ADT-LIA-nonlin**: Algebraic Data Types & Linear Integer Arithmetic - nonlinear clauses

[3] `https://tacas.info/toolympics2023.php`

[4] `https://www.starexec.org/`

[5] No solver requiring the syntactic restriction on the form of the clauses included in the LRA-TS track has been submitted in last two editions. Hence, as proposed in [15, 12], the LRA-TS and LRA-TS-par tracks have been discontinued. Similarly, by considering recent advances in solving techniques for CHCs including algebraic data types, the syntactic restriction on the constraints of the CHCs in the ADT-nonlin track, which requires to have all theory symbols encoded as ADTs (called "pure ADT" problems in [15]), was no longer needed. Hence, the track has been discontinued and replaced with a more general track combining LIA and ADTs (that is, ADT-LIA-nonlin).

In addition to the theories occurring in the above list (Linear Integer Arithmetic, Arrays, nonrecursive/recursive Algebraic Data Types, and combinations thereof), benchmarks in all tracks can also make use of the Bool theory.

Finally, in LIA constraints we allow the syntactic appearance of the function symbols $*$, *div*, *mod*, and *abs*. If these operations do appear, the benchmark is included/excluded from the set of LIA benchmarks according to the following rules: (i) if the second argument of any *div* and *mod* operation is not a constant term, the benchmark is excluded; (ii) if there is more than one non-constant term in any $*$ operation, the benchmark is excluded; (iii) otherwise, the operations are considered semantically linear and the benchmark is included.

2.2 Technical Resources

CHC-COMP 2023 was run, as well as in the previous editions, on the StarExec platform, but using different technical resources[12]. StarExec made available to the CHC community a queue, called `chc-seq.q`, consisting of 20 brand new nodes equipped with Intel(R) Xeon(R) Gold 6334 CPUs. The detailed specification of the machine is available on the StarExec webpage[6].

2.3 Test and Competition Runs

CHC solvers are evaluated by performing a *test* run and a *competition* run on the StarExec platform. A run involves submitting jobs to StarExec, that is, collections of ⟨solver-configuration, benchmark⟩ pairs.

The *test* run is used by the participants to get acquainted with the StarExec platform and test out their pre-submissions. Submitting a solver for *test* runs is optional. During this test phase, the organizers contact the participants if they find any issues with their submission so that the participants can fix it before their final submission. The participants are given a week in between the *test* and *competition* runs. In the *test* runs, a small set of randomly selected benchmarks is used, and each job is limited to 600s CPU time, 600s wall-clock time, and 64GB memory.

In *competition* runs, the final submissions of the solvers are evaluated to determine the outcome of the competition, that is, to rank the solvers that entered the competition. In these runs each job is limited to 1800s CPU time, 1800s wall-clock time, and 64GB memory.

Sometimes, the competition benchmarks expose soundness bugs in solvers. We catch these bugs if two solvers disagree on the satisfiability of a benchmark. At CHC-COMP, we keep things friendly by informing the participants about the inconsistency and giving them the benchmark to reproduce the issue. If we have time, we even give them a chance to fix the issue and resubmit their tool. If not, we disqualify the tool from the track.

The data gathered from the 'job information' CSV files produced by StarExec in the competition runs are used to rank the solvers. All 'job information' CSV files of the CHC-COMP 2023 runs are available on the StarExec space `CHC/CHC-COMP/CHC-COMP-23`[7].

2.4 Evaluation of the Competition Runs

The competing solvers were evaluated using the same approach as the 2022 edition [12].

[6] `https://www.starexec.org/starexec/public/machine-specs.txt`
[7] `https://www.starexec.org/starexec/secure/explore/spaces.jsp?id=538944`

The evaluation of the competition runs were done using the `summarize.py` script available at `https://github.com/chc-comp/scripts`; the script takes as input the 'job information' CSV file produced by StarExec at job completion, and produces a ranking of the solvers.

The ranking of solvers in each track is based on the score obtained by the solvers in the competition run for a track. The score is computed on the basis of the results provided by the solver on the benchmarks for that track. The result can be *sat*, *unsat*, or *unknown* (which includes solvers giving up, running out of resources, or crashing), and the score given by the number of *sat* or *unsat* results. In the case of ex-aequo, the ranking is determined by using the CPU time, which is the total CPU time needed by a solver to produce the results.

The tables in Appendix A also report in column '#unique' the number of *sat* or *unsat* results produced by a solver for benchmarks for which all other solvers returned *unknown*. The 'job information' files also include data about the space and memory consumption, which we consider less relevant and therefore are not reported in the tables (see also the CHC-COMP 2021 and CHC-COMP 2022 reports [15, 12]).

3 Benchmarks

3.1 Format

CHC-COMP accepts benchmarks in the SMT-LIB 2.6 format [2]. All benchmarks have to conform to the format described at `https://chc-comp.github.io/format.html`. This year, we updated the format to allow the declaration of ADTs using the `declare-datatypes` command. We support ADTs with any number of constructors and selectors as long as they are not parametric. Conformance is checked using the `format.py` script available at `https://github.com/chc-comp/scripts`.

3.2 Inventory

All benchmarks used for the competition are selected from repositories under `https://github.com/chc-comp`. Anyone can contribute benchmarks to this repository. This year, we got several new benchmarks for the track **ADT-LIA-nonlin**. Table 1 summarizes the number of benchmarks and unique benchmarks available in each repository. The organizers pick a subset of all available benchmarks for each year's competition. In the rest of this section, we explain the steps in this selection.

3.3 Processing Benchmarks

All benchmarks are processed using the `format.py` script, which is available at `https://github.com/chc-comp/scripts`. The command line for invoking the script is

```
> python3.9 format.py --out-dir <out-dir> --merge_queries True <smt-file>
```

The script attempts to put benchmark `<smt-file>` into CHC-COMP format. The `merge_queries` option merges multiple queries into a single query as discussed in previous editions of CHC-COMP [15]. In previous competitions, this script was not used in tracks containing ADTs because it did not print ADTs. This year, we updated the script to support printing ADTs in the SMT-LIB format using the `declare-datatypes` command. When printing ADTs are grouped as follows: if a constructor of ADT type *a* takes an argument of type ADT *b*, both *a* and *b* are grouped together. All ADTs in a group are declared together inside the same `declare-dataypes` command.

After formatting, benchmarks are categorized into one of the 6 competition tracks: LIA-lin, LIA-nonlin, LIA-lin-Arrays, LIA-nonlin-Arrays, ADT-LIA-nonlin, and LIA-nonlin-Arrays-nonrecADT. The scripts for categorizing benchmarks are available at `https://github.com/chc-comp/chc-tools/tree/master/format-checker`. This year, we added support for ADT tracks in the categorizing script. The script now checks for proper declaration of ADTs and proper usage of constructors, selectors, and recognizers. However, it does not check if a given ADT is recursive or not. Therefore, for the LIA-nonlin-Arrays-nonrecADT track, we manually verified that all ADTs are non-recursive. Benchmarks that could not be put in CHC-COMP compliant format and benchmarks that could not be categorized into any tracks are not used for the competition.

Repository	LIA-lin	LIA-nonlin	LIA-lin-Arrays	LIA-nonlin-Arrays	LIA-nonlin-Arrays-nonrecADT	ADT-LIA-nonlin
adtrem (*new*)						251/247
aeval	54/54					
aeval-unsafe	54/54					
chc-comp19			290/290			
eldarica-misc	149/136	69/66				
extra-small-lia	55/55					
hcai	101/87	133/131	39/39	25/25		
hopv	49/48	68/67				
jayhorn	75/73	7325/7224				
kind2		851/736				
ldv-ant-med			10/10	342/342		
ldv-arrays			3/2	821/546		
llreve	66/66	59/57	31/31			
quic3			43/43			
rust-horn (*new*)	11/11	6/6				56/56
seahorn	3379/2812	68/66				
solidity					2200/2174	
sv-comp	3150/2930	1643/1169	79/73	856/780		
synth/nay-horn		119/114				
synth/semgus				5371/4839		
tip-adt-lia (*new*)						320/320
tricera	405/405	4/4				
tricera/adt-arrays					156/156	
ultimate		8/8		23/23		
vmt	906/803					
total/**unique**	8454/**7534**	10353/**9648**	495/**488**	7438/**6555**	2356/**2330**	627/**623**

Table 1: Summary of benchmarks (total/unique).

3.4 Rating and Selection

This section describes the procedure used to select benchmarks for the competition.

We picked all unique benchmarks in the LIA-lin-Arrays track because of the scarcity of available benchmarks. In all other tracks, we followed a procedure similar to the past editions of the competition aiming at selecting a representative subset of the available benchmarks. In particular, we estimated how "easy" the benchmarks were and picked a mix of "easy" and "hard" instances. We say that a benchmark in a track is "easy" if it is solved by both the winner and the runner-up solvers in the corresponding track in CHC-COMP 2022, within a small time interval (30s).

Each benchmark was rated A/B/C based on how difficult the winner and the runner-up solvers found them. A rating of "A" is given if both solvers solved the benchmark, "B" if only one solver solved it, "C" if neither solved it, within the set timeout (30s). We ran all solvers using the same binaries and configurations submitted for CHC-COMP 2022.

Once we labeled each benchmark from a repository r, we decided the maximum number of instances, N_r, to take from the repository. N_r number was decided based on the total number of unique benchmarks and our knowledge about the benchmarks in repository r.

We picked at most $0.2 \cdot N_r$ benchmarks with rating A. Then, we picked at most $0.4 \cdot N_r$ benchmarks with rating B; namely, $0.2 \cdot N_r$ from those solved only by the winner solver and $0.2 \cdot N_r$ from those solved only by the runner-up solver. Finally, we picked at most $0.4 \cdot N_r$ benchmarks with rating C. If we did not find enough benchmarks with rating A, we picked the rest of the benchmarks with rating B (equally from those solved only by the winner and the runner-up). If we did not find enough benchmarks with rating B, we pick the remaining benchmarks from rating C.

This way, we obtained a mix of "easy" and "hard" benchmarks with a bias towards benchmarks that were not easily solved by either of the best solvers from the previous year's competition. The number of instances with each rating is given in Tables 2 and 3. The number of instances picked from each repository is given in Table 4. To pick `<num>` benchmarks of rating `<Y>`, we used the command

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

We were unable to run more than one solver for tracks containing ADTs (ADT-LIA-nonlin, LIA-nonlin-Arrays-nonrecADT). Only 3 solvers participated in tracks containing ADTs in CHC-COMP 2022: Spacer, Eldarica, and RInGen. RInGen does not support theories other than ADTs. The version of Eldarica submitted to CHC-COMP 2022 does not support the updated format of CHC-COMP 2023. Specifically, this version of Eldarica does not support the SMT-LIB syntax for recognizers[8]. Therefore, we were limited to using just one solver, Spacer, to select benchmarks for tracks containing ADTs. For each repository r, we decided a maximum number of instances N_r, ran Spacer on all benchmarks with the same timeout (30s), and picked $0.4 \cdot N_r$ benchmarks that Spacer solved (column B in Table 3) and $0.6 \cdot N_r$ benchmarks that Spacer did not solve (column C in Table 3).

The final set of benchmarks selected for CHC-COMP 2023 can be found in the github repository `https://github.com/chc-comp/chc-comp23-benchmarks`, and on StarExec in the public space `CHC/CHC-COMP-23/CHC-COMP-23-competition-runs`[9].

[8] since then, Eldarica has been updated to support recognizers. E.g. Eldarica v2.0.9 that participated in CHC-COMP 2023.

[9] `https://www.starexec.org/starexec/secure/explore/spaces.jsp?id=538230`

Repository	LIA-lin				LIA-nonlin				LIA-nonlin-Arrays			
	#A	#B		#C	#A	#B		#C	#A	#B		#C
		(*w*)	(*r*)			(*w*)	(*r*)			(*w*)	(*r*)	
aeval	12	9	4	29								
aeval-unsafe	17	0	12	25								
eldarica-misc	120	5	9	2	39	13	0	14				
extra-small-lia	30	13	8	4								
hcai	82	1	3	1	123	0	5	3	17	3	0	5
hopv	48	0	0	0	57	3	5	2				
jayhorn	73	0	0	0	3712	2275	1	1236				
kind2					650	70	0	16				
ldv-ant-med									0	128	0	214
ldv-arrays									7	195	0	344
llreve	61	0	5	0	48	4	2	3				
rust-horn	10	1	0	0	5	0	0	1				
seahorn	2089	65	69	589	60	1	2	3				
sv-comp	2854	1	74	1	1117	40	4	8	310	330	7	133
synth/nay-horn					70	20	4	20				
synth/semgus									737	2254	4	1844
tricera/svcomp20	43	7	4	351	4	0	0	0				
ultimate					0	1	0	7	0	0	0	23
vmt	711	31	7	54								
total	6150	133	195	1056	5885	2427	23	1313	1071	2910	11	2563

Table 2: The number of unique benchmarks with ratings A/B/C - Tracks: LIA-lin, LIA-nonlin, and LIA-nonlin-Arrays. B-rated benchmarks are reported in two sub-columns: (*w*) benchmarks solved only by the CHC-COMP 2022 winner, and (*r*) benchmarks solved only by the CHC-COMP 2022 runner-up solver.

Repository	LIA-nonlin-Arrays-nonrecADT		ADT-LIA-nonlin	
	#B	#C	#B	#C
adtrem			86	161
rust-horn			43	13
solidity	2109	65		
tip-adt-lia			39	281
tricera/adt-arrays	65	91		
total	2174	156	168	455

Table 3: The number of unique benchmarks with ratings B/C – Tracks: ADT-nonlin, and LIA-nonlin-Arrays-nonrecADT.

Repository	LIA-lin	LIA-nonlin	LIA-nonlin-Arrays	LIA-nonlin-Arrays-nonrecADT	ADT-LIA-nonlin
adtrem					125/125
aeval	30/30				
aeval-unsafe	30/30				
eldarica-misc	45/25	30/26			
extra-small-lia	30/22				
hcai	45/14	60/20	15/11		
hopv	30/6	30/16			
jayhorn	30/6	180/180			
kind2		90/52			
ldv-ant-med			60/60		
ldv-arrays			90/90		
llreve	30/11	45/18			
rust-horn					28/18
seahorn	90/90	45/15			
solidity				312/127	
sv-comp	90/38	90/48	135/135		
synth/nay-horn		60/48			
synth/semgus			135/135		
tip-adt-lia					160/160
tricera/svcomp20	60/60	3/0			
tricera/adt-arrays				156/122	
ultimate		6/5	15/15		
vmt	90/90				
total	600/**422**	639/**428**	450/**446**	468/**249**	313/**303**

Table 4: The number of benchmarks to select and the number of selected benchmarks from each repository.

4 Solvers

Seven solvers were submitted to CHC-COMP 2023: six competing solvers, and one solver *hors concours* (Spacer is co-developed by Hari Govind V K who is co-organizing the CHC-COMP 2023.).

Table 5 lists the submitted solvers together with the configurations used to run them on the competition tracks. Detailed descriptions of the solvers are provided in Section 7. The binaries of the solvers are available on the StarExec space `CHC/CHC-COMP/CHC-COMP-23-competitions-runs`.

Solver	LIA-lin	LIA-nonlin	LIA-lin-Arrays	LIA-nonlin-Arrays	LIA-nonlin-Arrays-nonrecADT	ADT-LIA-nonlin
Eldarica	`def`	`def`	`def`	`def`	`def`	`def`
Golem	`lia-lin`	`lia-nonlin`				
LoAT	`loat_horn`					
Theta	`fix`	`fix`	`fix`	`fix`		
Ultimate TreeAutomizer	`default`	`default`	`default`	`default`		
Ultimate Unihorn	`default`	`default`	`default`	`default`		
Spacer	`def`	`def`	`ARRAYS`	`ARRAYS`	`def`	`def`

Table 5: Solvers and configurations used in the tracks; an empty entry denotes that the solver did not enter the competition in that track. The configuration names have been taken as is from solver submissions.

5 Results

The results of the CHC-COMP 2023 are reported in Table 6. Detailed results are provided in Appendix A. All the data gathered from the execution of the StarExec jobs created for the competition run are available on the StarExec space `CHC/CHC-COMP/CHC-COMP-23-competitions-runs`.

	LIA-lin	LIA-nonlin	LIA-lin-Arrays	LIA-nonlin-Arrays	LIA-nonlin-Arrays-nonrecADT	ADT-LIA-nonlin
Winner	**Golem**	**Eldarica**	**Eldarica**	**Eldarica**	**Eldarica**	**Eldarica**
2nd place	Eldarica	Golem	Theta	Ultimate Unihorn		
3rd place	Theta	Ultimate Unihorn	Ultimate Unihorn	Theta		

Table 6: Results of CHC-COMP 2023. Spacer, which entered the competition as hors concours solver, placed in the first position of the LIA-lin, LIA-nonlin, LIA-lin-Arrays, and LIA-nonlin-Arrays tracks.

5.1 Observed Issues and Fixes during the Competition runs

This section describes the issues we have run across when using the tools entered in the competition and how we worked with the teams to overcome them.

Ultimate TreeAutomizer and Ultimate Unihorn Due to issues in building a version of Z3 that is able to run on StarExec, the final submission for the competition run of the solvers Ultimate TreeAutomizer and Ultimate Unihorn were completed on 14 April, 2023.

Theta In the competition runs of the LIA-nonlin-Arrays track we detected one inconsistent result: Theta (Theta-default in Table 7) reported *unsat* on one benchmark, while other solvers reported *sat*. The inconsistency was detected on April 14, and we informed the team on the same day by sending them the benchmark on which the issue was detected. The team submitted an updated version of Theta on April 15. Due to a configuration problem, the updated version of Theta reported *unknown* on all benchmarks. We informed the team on April 16, who provided an updated version of the solver (Theta-fix in Table 7) on the same day.

In the competition runs of the LIA-nonlin track we detected one inconsistent result: Theta-fix reported *sat*, while other solvers reported *unsat*. The Theta team was informed on April 19 by sending them the benchmark on which the issue was detected. The team submitted a fixed version (Thetafix-fix in Table 7) on April 19 that produced no inconsistent results.

The results presented in this report were produces using the fixed version. In Table 7 we report the results before and after the fixes.

Theta version	LIA-lin		LIA-nonlin		LIA-lin-Arrays		LIA-nonlin-Arrays	
	#sat	*#unsat*	*#sat*	*#unsat*	*#sat*	*#unsat*	*#sat*	*#unsat*
Theta-default	129	53	12	21	148	50	52	40
Theta-fix	121	49	9	20	135	50	45	39
Thetafix-fix	122	48	8	30	134	50	45	40

Table 7: Results produced by Theta before and after the fixes.

6 Conclusions and Final Remarks

We would like to congratulate the winners of the CHC-COMP 2023 (in alphabetical order): **Eldarica** (winner of the following tracks: LIA-nonlin, LIA-lin-Arrays, LIA-nonlin-Arrays, LIA-nonlin-Arrays-nonrecADT, and LIA-nonlin-Arrays), and **Golem** (winner of the LIA-lin track).

In organizing this edition of the competition we did our best to address some open issues discussed in the report of the CHC-COMP 2022 [12]. In particular, we have replaced the ADT-nonlin track with a more general track dealing with the combined theory of LIA and ADTs (ADT-LIA-nonlin), and we have extended the CHC format and the tools for processing and selecting the benchmarks to deal with ADTs. Moreover, as mentioned in the previous reports [15, 12], we have discontinued the obsolete tracks LRA-TS and LRA-TS-par. Finally, we have made a small change to the candidate benchmarks rating process by increasing the timeout used to evaluate their "hardness" (see Section 3.4). Ideally, we would have run the solvers with the same timeout as used in the competition (20 minutes). However, there are over 7500 benchmarks to pick from and we expect several timeouts irrespective of the time limit. Hence, for practical reasons, we set the timeout to 30 seconds for all solvers (previous editions had lower values that were dependent on the solver used to rate the benchmarks).

Below, we report the still open issues that should be further discussed for future editions, and the proposal for new tracks that emerged from the follow-up discussion we had after the presentation of the competition report at HCVS.

- **Validation of results** (also discussed in the previous editions [15, 12]). The ability of solvers to generate a witnesses (models or counter-examples) to support their results is a recurrent request by our community members. Several solvers have support for generating a witness. However, the witness is used mainly for debugging by the developers and having a common format for them is still a work in progress. As an additional issue, it is often the case that these witnesses are not for the original CHCs but for those obtained after many layers of pre-processing. Transforming these "internal" witnesses into a witness for the original problem is also a work in progress. While reaching a consensus on a common format for their encoding would require a thoughtful discussion involving all members of the CHC community, we could begin, as already proposed in the previous reports, by introducing in the CHC-COMP new tracks where the ability of producing a witness is taken into consideration in the computation of the score.
- **Status of benchmarks** (from [12]). In order to assess the correctness of the result provided by the solvers, each submitted benchmark should explicitly declare the expected result of the satisfiability problem. We propose to use the `( set-info` ⟨*keyword*⟩ ⟨*attr-value*⟩ `)` command with the `:status` as *keyword*, and either `sat` or `unsat` as *attr-value*.
- **Parallel tracks**. (Thanks to *Martin Blicha* for having sent us this note.) We propose a parallel version for each (or some) of the existing tracks. Instead of putting a limit on the CPU time, only a limit on the wall-clock time would be imposed in the parallel version. Parallel tracks can be implemented in two ways: either use the solvers' configuration submitted for the classical tracks, or allow a separate submission for the parallel tracks.

Finally, we would to stress once again that **a bigger set of benchmarks are needed**. Besides submitting their tools, all participants are invited to contribute with new benchmarks.

7 Solver Descriptions

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

7.1 Eldarica v2.0.9

Hossein Hojjat
University of Tehran, Iran

Philipp Rümmer
University of Regensburg, Germany and Uppsala University, Sweden

Algorithm. Eldarica [25] 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, bit-vectors, and the theory of heaps. 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 [34] as a library.

Configuration in CHC-COMP 2023. Eldarica is in the competition run with the option `-portfolio`, which enables a simple portfolio mode. Four instances of the solver are run in parallel, with the following options:

1. `-splitClauses:0 -abstract:off`,
2. `-splitClauses:1 -abstract:off -stac`,
3. `-splitClauses:1 -abstract:off`,
4. `-splitClauses:1 -abstract:relEqs` (the default options).

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

7.2 Golem

Martin Blicha
Università della Svizzera italiana, Switzerland

Konstantin Britikov
Università della Svizzera italiana, Switzerland

Algorithm. Golem is a CHC solver under active development that provides several backend engines implementing various SMT- and interpolation-based model-checking algorithms. It supports the theory of Linear Real or Integer Arithmetic and it is able to provide witnesses for both satisfiable and unsatisfiable CHC systems. Several back-end engines are implemeted in Golem:

- `lawi` is our re-implementation of the IMPACT algorithm [32]
- `spacer` is our re-implementation of the SPACER algorithm [28] and allows Golem to solve non-linear systems.
- `tpa` is our new model-checking algorithm based on doubling abstractions using Craig interpolants [7, 6].
- `bmc` implements the standard algorithm of Bounded Model Checking [4]
- `kind` implements a basic variant of k-induction [35]
- `imc` is our implementation of McMillan's first interpolation-based model-checking algorithm [31]

Architecture and Implementation. Golem is implemented in C++ and built on top of the interpolating SMT solver OPENSMT [26] which is used for both satisfiability solving and interpolation. The only dependencies are those inherited from OPENSMT: Flex, Bison and GMP libraries.

New Features in CHC-COMP 2023. Compared to the previous year, Golem has three new back-end engines: `bmc`, `kind` and `imc`. However, these engines support only transition systems and did not participate in the competition for this reason. Additionally, the preprocessing of the input system has improved significantly, without losing the ability to produce witnesses.

Configuration in CHC-COMP 2023. For LIA-nonlin track we used only `spacer` engine; the other engines cannot handle nonlinear system yet.

```
$ golem --engine spacer
```

For LIA-lin track, we used a trivial portfolio of `lawi`, `spacer` and `tpa` (in `split-tpa` mode) running independently.

```
$ golem --engine=spacer,lawi,split-tpa
```

`https://github.com/usi-verification-and-security/golem`
MIT LICENSE

7.3 LoAT chc-comp-2023

Florian Frohn
LuFG Informatik 2, RWTH Aachen University, Germany
Jürgen Giesl
LuFG Informatik 2, RWTH Aachen University, Germany

Algorithm. The *Loop Acceleration Tool* (LoAT) [18] is based on *Acceleration Driven Clause Learning* (ADCL) [19], a novel calculus for analyzing satisfiability of CHCs. LoAT's implementation of ADCL is based on a calculus for modular *loop acceleration* [17]. It can analyze linear Horn clauses over integer arithmetic. While ADCL can also prove satisfiability of CHCs, LoAT is currently restricted to proving unsatisfiability. Besides unsatisfiability of CHCs, LoAT can also prove non-termination and lower bounds on the worst-case runtime complexity of transition systems.

Architecture and Implementation. LoAT is implemented in C++. It uses the SMT solvers Z3 [33] and Yices [14], the recurrence solver PURRS [1], and the automata library libFAUDES [30].

New Features in CHC-COMP 2023. LoAT participates in the competition for the first time. Earlier version of LoAT could not analyze CHCs, but only transition systems.

Configuration in CHC-COMP 2023. At the competition, LoAT is run with the following arguments:

`--mode reachability` for proving reachability for transition systems or unsatisfiability of CHCs, respectively

`--format horn` for specifying that the input problem is given in the SMT-LIB-format for Horn clauses

`https://loat-developers.github.io/LoAT/`
GPL licence

7.4 Theta v4.2.3

Márk Somorjai

Mihály Dobos-Kovács

Levente Bajczi

András Vörös

Department of Measurement and Information Systems
Budapest University of Technology and Economics, Hungary

Algorithm. THETA decides the satisfiability of Constrained Horn Clauses by transforming it to a formal verification problem and employing an abstraction-based model checking technique. The input set of CHCs are transformed into a formal program representation named *Control Flow Automata (CFA)* [3] in a way that the unsatisfiability of the CHC problem is equivalent to the reachability of erroneous locations in the CFA. A bottom-up transformation is used for linear CHCs while a top-down transformation is done to nonlinear CHCs [36]. The erroneous state reachability of the created CFA is then checked using *CounterExample-Guided Abstraction Refinement (CEGAR)* [8], an iterative abstraction-based model checking algorithm.

Architecture and Implementation. THETA is a highly configurable model checking framework implemented in Java [21]. It supports various formalisms for the verification programs, engineering models and timed systems, among others. Verification is done by the main CEGAR engine, which utilizes SMT solvers through an SMTLIB interface to calculate interpolants and check the feasibility of paths. The CEGAR engine can be configured to use different abstraction domains and interpolation techniques. The framework offers a number of command line tools equipped with frontends that parse the input problem into a formalism. The bottom-up and top-down transformations from CHCs to CFA are implemented as a frontends for the `xcfa-cli` tool.

Configuration in CHC-COMP 2023. THETA is run with a sequential portfolio of 3 configurations listed below, using explicit value tracking, split predicate or cartesian predicate abstraction. Interpolation was set to backwards binary interpolation or sequential interpolation, calculated by Z3 [10] as the underlying SMT solver.

1. `--domain PRED_SPLIT --refinement BW_BIN_ITP --predsplit WHOLE`
2. `--domain PRED_CART --refinement BW_BIN_ITP --predsplit WHOLE`
3. `--domain EXPL --refinement SEQ_ITP`

THETA detects whether the input CHCs are linear or not and employs a bottom-up transformation for the former and a top-down transformation for the latter. The submitted Theta version and run scripts are available in the competition archive [37].

`https://github.com/ftsrg/theta`
Apache License 2.0

[10] `https://github.com/Z3Prover/z3`

7.5 Ultimate TreeAutomizer 0.2.3-dev-ac87e89

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
Frank Schüssele
University of Freiburg, Germany

Algorithm. The ULTIMATE TREEAUTOMIZER solver implements an approach that is based on tree automata [13]. 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[11] [24]. For checking satisfiability, we use the and Z3 SMT solver[12]. The tree automata are implemented in ULTIMATE's automata library[13]. The ULTIMATE framework is written in Java and build upon the Eclipse Rich Client Platform (RCP). The source code is available at GitHub[14].

Configuration in CHC-COMP 2023. 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://www.ultimate-pa.org/`
LGPLv3 with a linking exception for Eclipse RCP

[11] `https://ultimate.informatik.uni-freiburg.de/smtinterpol/`
[12] `https://github.com/Z3Prover/z3`
[13] `https://www.ultimate-pa.org/?ui=tool&tool=automata_library`
[14] `https://github.com/ultimate-pa/`

7.6 Ultimate Unihorn 0.2.3-dev-ac87e89

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

Frank Schüssele
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 [22] which implements an automata-based approach to software verification [23].

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[15]and Z3[16]. The ULTIMATE framework is written in Java and build upon the Eclipse Rich Client Platform (RCP). The source code is available at GitHub[17].

Configuration in CHC-COMP 2023. 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 `chccomp-Unihorn_Default.epf` settings file. Both files can be found in toolchain (resp. settings) folder of ULTIMATE's repository.

`https://www.ultimate-pa.org/`
LGPLv3 with a linking exception for Eclipse RCP

[15]`https://ultimate.informatik.uni-freiburg.de/smtinterpol/`
[16]`https://github.com/Z3Prover/z3`
[17]`https://github.com/ultimate-pa/`

References

[1] Roberto Bagnara, Andrea Pescetti, Alessandro Zaccagnini & Enea Zaffanella (2005): *PURRS: Towards Computer Algebra Support for Fully Automatic Worst-Case Complexity Analysis*, doi:10.48550/arXiv.cs/0512056.

[2] Clark Barrett, Pascal Fontaine & Cesare Tinelli (2016): *The Satisfiability Modulo Theories Library (SMT-LIB)*. www.SMT-LIB.org.

[3] Dirk Beyer & M. Erkan Keremoglu (2011): *CPAchecker: A Tool for Configurable Software Verification*. In Ganesh Gopalakrishnan & Shaz Qadeer, editors: *Computer Aided Verification*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 184–190, doi:10.1007/978-3-642-22110-1_16.

[4] 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, doi:10.1007/3-540-49059-0_14.

[5] Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan & Andrey Rybalchenko (2015): *Horn Clause Solvers for Program Verification*. In Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner & Wolfram Schulte, editors: *Fields of Logic and Computation II: Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, Springer International Publishing, Cham, pp. 24–51, doi:10.1007/978-3-319-23534-9_2.

[6] Martin Blicha, Grigory Fedyukovich, Antti E. J. Hyvärinen & Natasha Sharygina (2022): *Split Transition Power Abstractions for Unbounded Safety*. In Alberto Griggio & Neha Rungta, editors: *Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design – FMCAD 2022*, TU Wien Academic Press, pp. 349–358.

[7] Martin Blicha, Grigory Fedyukovich, Antti E. J. Hyvärinen & Natasha Sharygina (2022): *Transition Power Abstractions for Deep Counterexample Detection*. In Dana Fisman & Grigore Rosu, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer International Publishing, Cham, pp. 524–542, doi:10.1007/978-3-030-99524-9_29.

[8] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu & Helmut Veith (2003): *Counterexample-Guided Abstraction Refinement for Symbolic Model Checking*. *J. ACM* 50(5), p. 752–794, doi:10.1145/876638.876643.

[9] Emanuele De Angelis, Fabio Fioravanti, John P. Gallagher, Manuel V. Hermenegildo, Alberto Pettorossi & Maurizio Proeitti (2021): *Analysis and Transformation of Constrained Horn Clauses for Program Verification*. *Theory and Practice of Logic Programming*, p. 1–69, doi:10.1017/S1471068421000211.

[10] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi & Maurizio Proietti (2014): *VeriMAP: A Tool for Verifying Programs through Transformations*. In: *Tools and Algorithms for the Construction and Analysis of Systems, TACAS '14*, Lecture Notes in Computer Science 8413, Springer, pp. 568–574, doi:10.1007/978-3-642-54862-8_47.

[11] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi & Maurizio Proietti (2022): *Satisfiability of constrained Horn clauses on algebraic data types: A transformation-based approach*. *J. Log. Comput.* 32(2), pp. 402–442, doi:10.1093/logcom/exab090.

[12] Emanuele De Angelis & Hari Govind V K (2022): *CHC-COMP 2022: Competition Report*. *Electronic Proceedings in Theoretical Computer Science* 373, pp. 44–62, doi:10.4204/eptcs.373.5.

[13] Daniel Dietsch, Matthias Heizmann, Jochen Hoenicke, Alexander Nutz & Andreas Podelski (2019): *Ultimate TreeAutomizer (CHC-COMP Tool Description)*. In Emanuele De Angelis, Grigory Fedyukovich, Nikos Tzevelekos & Mattias Ulbrich, editors: *Proceedings of the Sixth Workshop on Horn Clauses for Verification and Synthesis and Third Workshop on Program Equivalence and Relational Reasoning, HCVS/PERR@ETAPS 2019, Prague, Czech Republic, 6-7th April 2019*, *EPTCS* 296, pp. 42–47, doi:10.4204/EPTCS.296.7.

[14] Bruno Dutertre (2014): *Yices 2.2*. In Armin Biere & Roderick Bloem, editors: *CAV '14*, Springer International Publishing, pp. 737–744, doi:10.1007/978-3-319-08867-9_49.

[15] Grigory Fedyukovich & Philipp Rümmer (2021): *Competition Report: CHC-COMP-21*. In Hossein Hojjat & Bishoksan Kafle, editors: *Proceedings 8th Workshop on Horn Clauses for Verification and Synthesis, HCVS@ETAPS 2021, Virtual, 28th March 2021*, *EPTCS* 344, Open Publishing Association, pp. 91–108, doi:10.4204/EPTCS.344.7.

[16] Grigory Fedyukovich, Yueling Zhang & Aarti Gupta (2018): *Syntax-Guided Termination Analysis*. In Hana Chockler & Georg Weissenbacher, editors: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, *Lecture Notes in Computer Science* 10981, Springer, pp. 124–143, doi:10.1007/978-3-319-96145-3_7.

[17] Florian Frohn (2020): *A Calculus for Modular Loop Acceleration*. In Armin Biere & David Parker, editors: *TACAS '20*, Springer International Publishing, pp. 58–76, doi:10.1007/978-3-030-45190-5_4.

[18] Florian Frohn & Jürgen Giesl (2022): *Proving Non-Termination and Lower Runtime Bounds with LoAT (System Description)*. In Jasmin Blanchette, Laura Kovács & Dirk Pattinson, editors: *IJCAR '22*, Springer International Publishing, pp. 712–722, doi:10.1007/978-3-031-10769-6_41.

[19] Florian Frohn & Jürgen Giesl (2023): *ADCL: Acceleration Driven Clause Learning for Constrained Horn Clauses*, doi:10.48550/arXiv.2303.01827.

[20] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea & Andrey Rybalchenko (2012): *Synthesizing software verifiers from proof rules*. In Jan Vitek, Haibo Lin & Frank Tip, editors: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, ACM, pp. 405–416, doi:10.1145/2254064.2254112.

[21] Ákos Hajdu & Zoltán Micskei (2020): *Efficient Strategies for CEGAR-Based Model Checking*. *Journal of Automated Reasoning* 64(6), pp. 1051–1091, doi:10.1007/s10817-019-09535-x.

[22] Matthias Heizmann, Max Barth, Daniel Dietsch, Leonard Fichtner, Jochen Hoenicke, Dominik Klumpp, Mehdi Naouar, Tanja Schindler, Frank Schüssele & Andreas Podelski (2023): *Ultimate Automizer and the CommuHash Normal Form - (Competition Contribution)*. In Sriram Sankaranarayanan & Natasha Sharygina, editors: *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II*, *Lecture Notes in Computer Science* 13994, Springer, pp. 577–581, doi:10.1007/978-3-031-30820-8_39.

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

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

[25] Hossein Hojjat & Philipp Rümmer (2018): *The ELDARICA Horn Solver*. In: *2018 Formal Methods in Computer Aided Design, FMCAD*, pp. 1–7, doi:10.23919/FMCAD.2018.8603013.

[26] 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, doi:10.1007/978-3-319-40970-2_35.

[27] Bishoksan Kafle, John P. Gallagher & José F. Morales (2016): *RAHFT: A Tool for Verifying Horn Clauses Using Abstract Interpretation and Finite Tree Automata*. In Swarat Chaudhuri & Azadeh Farzan, editors: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, *Lecture Notes in Computer Science* 9779, Springer, pp. 261–268, doi:10.1007/978-3-319-41528-4_14.

[28] Anvesh Komuravelli, Arie Gurfinkel & Sagar Chaki (2016): *SMT-based Model Checking For Recursive Programs*. *Formal Methods in System Design* 48(3), pp. 175–205, doi:10.1007/s10703-016-0249-4.

[29] 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 International Conference on Programming Language Design and Implementation*, PLDI 2021, ACM, p. 451–465, doi:10.1145/3453483.3454055.

[30] *libFAUDES Library*. Available at `https://fgdes.tf.fau.de/faudes/index.html`.

[31] Kenneth L. McMillan (2003): *Interpolation and SAT-Based Model Checking*. In Warren A. Hunt & Fabio Somenzi, editors: *Computer Aided Verification*, Springer, Berlin Heidelberg, pp. 1–13, doi:10.1007/978-3-540-45069-6_1.

[32] 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, doi:10.1007/11817963_14.

[33] Leonardo de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In C. R. Ramakrishnan & Jakob Rehof, editors: *TACAS '08*, Springer Berlin Heidelberg, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[34] 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.

[35] Mary Sheeran, Satnam Singh & Gunnar Stålmarck (2000): *Checking Safety Properties Using Induction and a SAT-Solver*. In Warren A. Hunt & Steven D. Johnson, editors: *Formal Methods in Computer-Aided Design*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 127–144, doi:10.1007/3-540-40922-X_8.

[36] Márk Somorjai, Mihály Dobos-Kovács, Zsófia Ádám, Levente Bajczi & András Vörös (2023): *Bottoms Up for CHCs: Novel Transformation of Linear Constrained Horn Clauses to Software Verification*. *Electronic Proceedings in Theoretical Computer Science*.

[37] Márk Somorjai, Mihály Dobos-Kovács, Levente Bajczi & András Vörös (2023): *Tool Archive of Theta for CHC-COMP 2023*, doi:10.5281/zenodo.7954684.

[38] 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*, Springer International Publishing, Cham, pp. 367–373, doi:10.1007/978-3-319-08587-6_28.

[39] Hiroshi Unno, Tachio Terauchi & Eric Koskinen (2021): *Constraint-Based Relational Verification*. In Alexandra Silva & K. Rustan M. Leino, editors: *Computer Aided Verification*, Springer International Publishing, Cham, pp. 742–766, doi:10.1007/978-3-030-81685-8_35.

[40] Weikun Yang, Grigory Fedyukovich & Aarti Gupta (2019): *Lemma Synthesis for Automating Induction over Algebraic Data Types*. In Thomas Schiex & Simon de Givry, editors: *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings*, *Lecture Notes in Computer Science* 11802, Springer, pp. 600–617, doi:10.1007/978-3-030-30048-7_35.

A Detailed results

Solver	Score	#sat	#unsat	CPU time/s	Wall-clock/s	#unique
Spacer	265	199	66	274397	138310	43
Golem	229	148	81	368980	129633	8
Eldarica	219	160	59	385851	112832	23
Theta	170	122	48	426006	370425	0
U. Unihorn	103	72	31	449683	384389	0
U. TreeAutomizer	81	50	31	537858	517349	0
LoAT	50	0	50	287878	287841	4

Table 8: Solver performance on LIA-lin track

Solver	Score	#sat	#unsat	CPU time/s	Wall-clock/s	#unique
Spacer	384	235	149	90842	50781	38
Eldarica	330	185	145	218944	79522	9
Golem	310	178	132	248569	248578	3
U. Unihorn	121	72	49	470768	389915	0
Theta	38	8	30	687374	666145	0
U. TreeAutomizer	34	5	29	569895	531158	0

Table 9: Solver performance on LIA-nonlin track

Solver	Score	#sat	#unsat	CPU time/s	Wall-clock/s	#unique
Spacer	281	212	69	359439	187454	81
Eldarica	220	150	70	478284	166185	15
Theta	184	134	50	285884	271624	0
U. Unihorn	164	122	42	242113	206799	1
U. TreeAutomizer	131	96	35	239591	229783	0

Table 10: Solver performance on LIA-lin-Arrays track

Solver	Score	#sat	#unsat	CPU time/s	Wall-clock/s	#unique
Spacer	258	148	110	290925	156914	75
Eldarica	206	122	84	454921	184851	26
U. Unihorn	96	37	59	234519	199416	0
Theta	85	45	40	588095	569760	4
U. TreeAutomizer	56	6	50	276025	250747	0

Table 11: Solver performance on LIA-nonlin-Arrays track

Solver	Score	#sat	#unsat	CPU time/s	Wall-clock/s	#unique
Eldarica	176	85	91	114521	42212	57
Spacer	120	59	61	195321	107046	1

Table 12: Solver performance on LIA-nonlin-Arrays-nonrecADT track

Solver	Score	#sat	#unsat	CPU time/s	Wall-clock/s	#unique
Eldarica	58	22	36	433561	150012	30
Spacer	30	3	27	440259	290358	2

Table 13: Solvers performance on ADT-LIA-nonlin track