Theta as a Horn Solver

Levente Bajczi◉   Milán Mondok◉   Vince Molnár◉

Department of Artificial Intelligence and Systems Engineering
Faculty of Electrical Engineering and Informatics
Budapest University of Technology and Economics
Műegyetem rkp. 3., H-1111 Budapest, Hungary.

`{bajczi,mondok,molnarv}@mit.bme.hu`

THETA is a verification framework that has participated in the CHC-COMP competition since 2023. While its core approach – based on transforming constrained Horn clauses (CHCs) into control-flow automata (CFAs) for analysis – has remained mostly unchanged, THETA's verification techniques, design trade-offs, and limitations have remained mostly unexplored in the context of CHCs. This paper fills that gap: we provide a detailed description of the algorithms employed by Theta, highlighting the unique features that distinguish it from other CHC solvers. We also analyze the strengths and weaknesses of the tool in the context of CHC-COMP benchmarks. Notably, in the 2025 edition of the competition, Theta's performance was impacted by a configuration issue, leading to suboptimal results. To provide a clearer picture of Theta's actual capabilities, we re-execute the tool on the competition benchmarks under corrected settings and report on the resulting performance.

1   Introduction

Constrained Horn clauses (CHCs) have emerged as a standard intermediate representation for various verification tasks (such as program- or smart-contract verification [17, 24]) and are the focus of the annual CHC-COMP competition[13]. Over the past three years, the *Theta* framework[1], an open-source, modular model checker developed at the Critical Systems Research Group, Budapest University of Technology and Economics [29, 18] has participated as a CHC solver in CHC-COMP, aiming to adapt techniques from software model checking to solve constrained horn clauses.

Originally conceived as a framework for predicate abstraction with configurable refinement strategies, Theta's CHC-solving capabilities are built upon a pre-analysis transformation that converts CHC systems into control-flow automata (CFAs), enabling the reuse of mature abstraction-based algorithms, and more novel, bounded techniques [5]. This transformation step has been described in an earlier publication [26], but beyond that, there has been no comprehensive documentation of how Theta handles CHC inputs, nor an in-depth assessment of its effectiveness across different categories of CHC problems.

This paper fills that gap. We describe in detail the architecture of Theta, and its main algorithms – including abstraction-refinement and bounded techniques – including the sequential portfolio we use for automated verification, with next to no need for user input when choosing a performant algorithm. We also analyze how well these methods scale and where they fall short, both in theory and in practice.

---

[1] `https://github.com/ftsrg/theta`

Submitted to:
HCVS 2025

We also reflect on Theta's 2025 CHC-COMP submission, whose performance was diminished by a misconfiguration that resulted in the first step of the portfolio never advancing to others. While the competition results for this year are already finalized, they do not accurately represent the tool's capabilities. To address this, we rerun Theta on the official benchmark set under corrected settings and present a comparative analysis of the results.

The remainder of the paper is structured as follows. In Section 2, we describe Theta's CHC-solving approach in detail. Section 3 presents the overall software architecture of the tool, including its modular design and extensibility. Section 4 evaluates the key strengths and limitations of Theta's approach, informed by our experience in CHC-COMP and beyond. Section 5 outlines the specific configuration and setup used in the 2025 CHC-COMP submission, highlighting the cause of the performance issues and detailing our re-evaluation methodology. Finally, Section 6 provides information on the availability of the Theta tool, its source code, and the data used in our experiments to support reproducibility.

2   Verification Approach

Theta supports multiple verification techniques for analyzing software systems encoded as Constrained Horn Clauses (CHCs). These techniques are grounded in symbolic model checking and operate over control flow automata (CFA), which are derived via a structural transformation from the original CHC system [26]. This section outlines the main verification approaches implemented in Theta, with a focus on the ones used in the CHC-COMP configuration.

2.1   CHC-to-CFA Transformation

As detailed in prior publications [26, 3], a CHC problem can be transformed to a program-like structure – namely, a control flow automaton (CFA) [18] with variables $V = \{v_1, v_2, \ldots, v_n\}$ over domains $D_{v_1}, D_{v_2}, \ldots, D_{v_n}$, locations L, and edges $E \subseteq L \times Ops \times L$, where Ops can have:

- *assume(expr)*,
- *assign(expr$_{lhs}$, expr$_{rhs}$)*, and
- *havoc($v \in V$)*

instructions for guards, variable updates, and nondeterministic value assignments, respectively. Domains of variables are subsets of domains in SMT.

Multi-procedure programs can be represented as a set of named CFAs, where procedures can be called as part of an expression. We assume all procedures are side-effect free and behave like mathematical functions (a sane assumptions, given they were transformed from the uninterpreted predicates of a CHC problem).

With a *forward*, or *bottom-up* transformation [26], the resulting artifact is always represented as a single CFA, i.e., only only a single procedure will exist, and no function invocations exist in the expressions.

With a *backward*, or *top-down* transformation [26], the result may be multiple CFAs, each invoking zero or more other CFA procedures as functions. In the case of non-linear clauses, only this method can be used.

2.2   CEGAR-based Verification

Theta's core verification engine is based on the Counterexample-Guided Abstraction Refinement (CEGAR) paradigm [12], where verification is performed by iteratively refining an abstract model of the
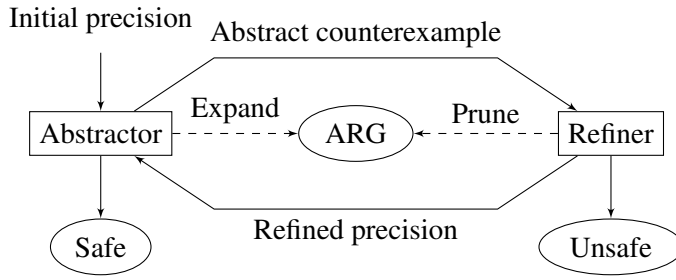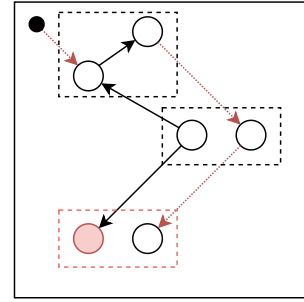
Figure 1: The CEGAR loop

Figure 2: An ARG

system. The main data structure maintained during this process is the *Abstract Reachability Graph* (ARG) [8], which over-approximates the concrete reachable state space.

In the ARG, abstract states represent sets of concrete states. This abstraction ensures soundness: if a concrete error state is reachable, then so is an abstract error state. However, the converse is not guaranteed, leading to potential *spurious counterexamples*. Figure 2 illustrates this: an error state may appear reachable within the abstract graph, but it may not correspond to a concrete execution path, prompting refinement.

A key mechanism in ARG-based abstraction is *state covering*. If an abstract state s_1 is logically implied by another state s_2 – for example, if s_1 includes constraints $a = 2 \land b = 3$ and s_2 includes only $a = 2$ – then s_1 is *covered* by s_2. This allows the search to avoid exploring redundant paths, improving scalability. This is particularly useful in programs with loops that do not influence the error condition: intermediate states can be covered by the loop header, avoiding repeated exploration.

The CEGAR process operates in a loop, as shown in Figure 1. The loop consists of two main components: the *abstractor* and the *refiner*. The abstractor constructs or expands the ARG based on the current abstraction precision. It checks whether a (possibly spurious) error state is reachable. If no such state is reachable, the system is proven *safe*, since the abstraction is conservative.

If an error state is found, the abstractor extracts one or more *abstract counterexamples*-paths in the ARG leading to the error. These are handed to the refiner, which first checks whether the path is feasible in the concrete system. If a feasible path is found, the program is *unsafe*. Otherwise, the refinement process strengthens the abstraction by computing a more precise abstraction (typically by deriving new predicates) and pruning the ARG to the point of infeasibility. The refined abstraction is then passed back to the abstractor, and the cycle continues.

The CEGAR loop described so far is a high-level specification focusing on outcomes rather than implementation details. This reflects CEGAR's inherent modularity: its components can be freely interchanged as long as they fulfill compatible roles.

THETA's CEGAR implementation emphasizes modularity, offering multiple configurable components. Among these, the two most critical are the *abstract domain* and the *refinement algorithm* (Implementation details not directly relevant to this paper are documented in [18]).

2.2.1 Abstract Domain

The abstract domain defines the abstraction's foundation. THETA supports two main domains: the *explicit value* domain and the *predicate* domain. The predicate domain includes variants such as cartesian,

boolean, and split boolean abstractions.

Formally, an abstract domain is a tuple $D = (S, \top, \bot, \sqsubseteq, expr)$, where:
- S is a lattice of abstract states,
- $\top \in S$ is the top (most abstract) element,
- $\bot \in S$ is the bottom (contradictory) element,
- $\sqsubseteq$ is the partial order on S,
- *expr* maps an abstract state to its corresponding concrete expression.

Mapping these concepts to the two domains, we get:

Explicit Domain.   Tracks a set of variables explicitly:
- S: A variable assignment of each *tracked* variable to a value of its domain, extended with top (arbitrary value) and bottom (no assignment possible) elements.
- $\top \in S$: No specific value is assigned to any of the tracked variables.
- $\bot \in S$: No assignment is possible to the tracked variables.
- $\sqsubseteq \subseteq S \times S$: $(s_1 \in S) \sqsubseteq (s_2 \in S) \iff (s_1 = s_2) \vee (s_1 = \bot) \vee (s_2 = \top)$
- *expr*: The conjunction of the equality expressions for each tracked variable and their value

In control-flow automata (CFA), locations are always explicitly tracked to maintain a one-to-many mapping between locations and abstract states. To mitigate state explosion, a *maxenum* parameter limits value enumeration per step. For example, when a variable takes nondeterministic 32-bit values, the explicit domain keeps it at $\top$ rather than enumerating all possibilities, enabling efficient safety checks. This makes the CEGAR loop potentially incomplete, but we can detect a non-progressing refinement cycle, and stop the analysis.

Predicate Domain.   Tracks a set of boolean predicates, such that S is a Boolean combination of first-order logic (FOL) predicates. $\top \in S$ is *True* and $\bot \in S$ is *False*. The partial order $\sqsubseteq \subseteq S \times S$ is the logical implication (i.e., $(s_1 \in S) \sqsubseteq (s_2 \in S) \iff (s_1 \implies s_2)$). *expr* is the conjunction of the predicates.

A special version of the predicate domain is Cartesian predicate abstraction [6]. Here, only conjunction of ponated and negated FOL predicates are allowed in the states, as opposed to arbitrary boolean combinations of predicates.

2.2.2   Refinement Algorithm

Refinement either incorporates predicates from the initial precision or extracts them while refuting infeasible abstract counterexamples, gradually guiding the abstraction toward sufficient precision for verification. Two main strategies exist in THETA:
- **Single-counterexample refinement:** Refines precision based on one counterexample at a time, using binary [21] or sequence interpolation [18].
- **Multi-counterexample refinement:** Uses all counterexamples from the ARG for a combined refinement, using tree interpolation [1].

The refinement strategies, if the trace is found to be spurious, return a *refutation*, containing the cause of the contradiction on the path. This information is used to update the abstraction precision.

2.3   Bounded, Property-Directed and Decision-Diagram-Based Techniques

The second major family of techniques is implemented in the EMERGENTHETA configuration [5]. This includes bounded model checking (BMC), *k*-induction, interpolation-based model checking (IMC),

property-directed reachability (PDR/IC3), and (generalized) saturation. These algorithms are defined on the *symbolic transition system* (STS) formalism, which describe safety (reachability) problems using 3 SMT formulas (I, T, P for *initial states*, *transition relation* and *safety property*, respectively). For example, the STS $I : x = 0$, $T : x' = x + 1$, $P : x < 5$ describes a state space where the x variable has the initial value 0, x gets incremented by 1 when a transition fires and the safety property states that in all safe states x is lower than 5. The CFAs created from CHC problems are transformed into STS by encoding the control location as a data variable and transforming the operations to characteristic functions. EMERGENTHETA also supports chainable STS-to-STS transformations that can be used to encode enhanced analyses into STS problems and thus allow for *reversed exploration*, *implicit predicate abstraction* [28] and *liveness checking* [10]. Our CFA-to-STS transformation currently only supports single procedure CFAs, which means that the algorithms of the EMERGENTHETA configuration can only be used with the *forward* [26] CHC-to-CFA mapping.

2.3.1 Bounded Techniques

The bounded techniques of EMERGENTHETA focus on finding bugs or constructing inductive invariants within bounded or unrolling-based search spaces. Bounded model checking [11] (with a loop-free check to detect finite state spaces) checks for violations of the safety property up to a finite iteratively incremented bound. *K*-induction [25] and interpolation-based model checking [20] can complement BMC with additional checks that attempt to prove that the model is safe. In case of a safe verdict, IMC can provide an (overapproximating) inductive invariant that can be used to construct a model for the original CHC problem.

2.3.2 Property-Directed Reachability

Property-directed reachability [16] analyzes the model incrementally through the proof of several lemmas, which eventually form an inductive invariant proving the system correct or direct the search towards a counterexample. Initially designed for hardware model-checking, this algorithm can efficiently handle Boolean variables, but requires abstraction to reason about integer domains. Our current implementation of PDR uses STS as input and does not exploit the structural information present in CFAs. We also have an experimental implementation of the "two-dimensional" IC3 algorithm described in [19], which handles structural knowledge present in CFAs explicitly, but this configuration is not yet stable enough to be included in the portfolio for CHCs.

2.3.3 Decision-Diagram-Based Techniques

Decision diagrams [22] offer a compact way to represent sets of vectors. Substitution diagrams [23] (see Fig. 3) allow us to build multi-valued decision diagrams (MDD) from the SMT formulas of the STS models in a top-down manner, without the need to explicitly enumerate all possible valuations beforehand. For an STS model with k variables, substitution diagrams with k levels are used to represent the initial states and the safety property, while a diagram with $2k$ levels is used to characterize the transition relation. *Generalized saturation* (GSAT) [22] can be used to enumerate the state space characterized by decision diagrams in a manner that decomposes exploration into smaller ones on submodels exploiting the locality of transitions. The saturation algorithm constructs a decision diagram describing all reachable states of the system, which can be used as a basis for model generation of the original CHC problem. A current limitation of this approach is that it can only handle finite state spaces, which we plan to alle-

viate via abstraction. We currently cannot wrap the saturation algorithm in a CEGAR loop with implicit predicate abstraction, because it does not yet yield a diagnostic trace for unsafe models.
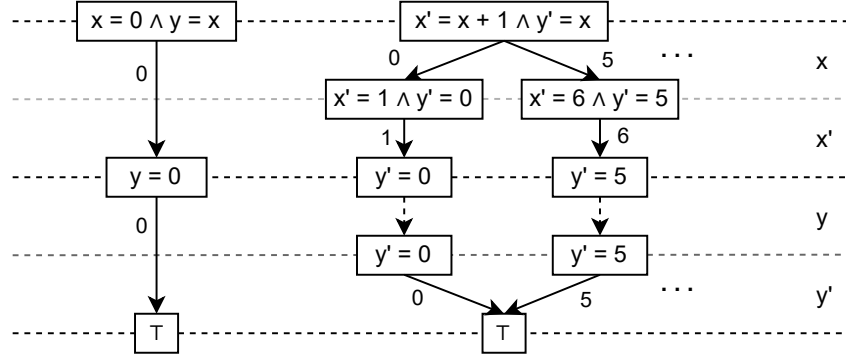

Figure 3: Substitution diagram characterizing the initial states $I : x = 0 \land y = x$ (left) and 2k-level substitution diagram characterizing the transition relation $T : x' = x + 1 \land y' = x$ (right). Each level has an associated variable, nodes correspond to SMT formulas, while the edges represent substituting the variable of the current level with a literal value in the SMT formula of the source node. We omit edges leading to the terminal $\bot$ node. The diagram on the right has infinitely many edges leaving the top node, we only display 0 and 5 for readability.

2.4   CHC-Solvers as Backend

Theta also includes an experimental configuration known as THORN, which integrates external CHC solvers to discharge verification conditions or analyze the CHC system directly. Because we feel this would be unfair to the external CHC solvers, who are participants of CHC-COMP themselves, we do not use the THORN configuration at CHC-COMP.

2.5   Portfolio Strategy

For CHC-COMP, Theta employs a sequential-portfolio-based approach, combining the CEGAR and bounded verification techniques. This allows us to balance thoroughness and efficiency, applying lightweight methods where appropriate while falling back on abstraction refinement when deeper reasoning is needed. This design enables Theta to handle a broad spectrum of CHC tasks, from shallow counterexample detection to complex proofs requiring inductive invariants.

At CHC-COMP'25, we use a sequential combination of the following configurations:
1. Bounded model checking (BMC),
2. K-induction (kIND),
3. Interpolation-based model checking (IMC),
4. Generalized saturation (GSAT),
5. Boolean predicate abstraction with backwards-binary interpolation (BOOL),
6. Cartesian predicate abstraction with backwards-binary interpolation (CART),
7. Explicit-value abstraction with sequential interpolation (EXPL),
8. Fallback configuration with our SV-COMP'25 portfolio [27].

The exact order of the first 7 configurations depend on the arithmetic of the task (based on empirical experience), and the first 3 may only be used with linear clauses (as these only support the *backward* transformation [26]). For all configurations, we use either Z3 [14] 4.3.0 (which is outdated, but interpolates much better than the newer versions of Z3), or CVC5 [7] 1.0.8.

While all configurations feature a timeout value, we also handle other sources of premature completion of incomplete configurations, such as SMT solver issues, or runtime exceptions.

Even though there are many potential tweaks we could make to this portfolio to marginally enhance its performance, such as changing SMT solvers dynamically, we do not believe these provide enough advantages to overcome the disadvantage of a more fragmented, and harder to maintain portfolio.

2.6 Model Generation

While all configurations mentioned in Sect. 2.5 can prove safety, not all of them provide an easy-to-interpret proof that can be converted into a model for the original CHC problem. However, a few (namely: EXPL, BOOL, CART, IMC and GSAT) already provide an overabstraction of the state space as a formula, and thus, can be transformed into a model easily.

Because of our pre-processing step transforming CHCs to CFA, we can rely on a known mapping from locations to CHC predicates. Given this knowledge, we construct a formula for all variables in the program for each location, then – using our knowledge of the original predicate – extract which variables are of interest, and existentially quantify the rest.

As an example, if a CFA has the variables $\{x,y,z\}$, but a predicate only takes a single argument x: $inv(x)$, then a formula $(loc = inv) \implies x = y+2 \wedge y = z+1$ will be transformed into the model

$$\forall x : inv(x) \iff \exists y,z : x = y+2 \wedge y = z+1$$

While this could be further simplified in some cases, we do not (yet) perform any simplifications on our models. Furthermore, we do not yet output a trace or counterexample for the `unsat` case.

3 Software Architecture

THETA is built on a modular, layered architecture that separates concerns across four key layers: *frontends*, *formalisms*, *algorithms*, and *SMT solvers*. This architecture enables flexibility, reuse, and easy experimentation with different verification techniques and input formats.

3.1 Frontends

The frontend layer is responsible for parsing and preprocessing the input model. For CHC solving, we use our own custom parser for the SMT-LIB V2 format, designed specifically to support Horn clauses. This parser converts logical formulae into an internal intermediate representation that supports programmatic analysis and transformation.

3.2 Formalisms

In THETA, all verification algorithms operate on an internal formalism. For CHC programs, this is an *extended control-flow automaton* (XCFA), which augments classical CFAs with additional features such as procedure calls and threads. When using certain techniques such as bounded- or saturation-based

analyses, the XCFA is further translated into a symbolic transition system that enables efficient state space exploration using only logical formulae.

3.3 Algorithms

The architecture supports a range of verification algorithms. The algorithm layer operates independently of the input language and solver, relying only on the abstract view provided by the formalism layer. This design enables algorithm reuse across frontends and verification domains.

3.4 SMT Solvers

The solver layer provides decision procedures for logical queries during verification. THETA can use Z3 [14] via its Java API, any solver compliant with the SMT-LIB v2 standard [15], or a wide selection of up-to-date solvers through *JavaSMT* [2] – an abstraction layer over modern SMT solvers such as Z3, CVC4, CVC5, MathSAT, SMTInterpol, and Boolector.

Overall, the layered design of THETA allows each component to evolve independently and be extended with minimal effort, which has enabled rapid development of CHC solving capabilities on top of the existing software verification infrastructure.

4 Strengths and Weaknesses of the Approach

While THETA has been a participant at CHC-COMP for 3 years now, the main focus of it has not been to solve CHCs, but to show how a general-purpose verification framework fares compared to dedicated CHC solvers with a light-weight pre-processing step [26].

This lack of dedication led to an oversight when assembling the release for CHC-COMP'25. We omitted a very important flag that allows THETA to call itself as a subprocess, thus allowing precise time- and memory-usage control. Without this option, the first configuration of the sequential portfolios were running without time limits, and thus, the whole portfolio's performance was severely hurt. We often use a lightweight (and therefore quick) but quite underpowered configuration for starting our portfolio in the case of linear tasks, the supposed strength of THETA. As we did not uncover this problem in time for submitting the final version, unfortunately, we finished quite behind other competitors in most of the categories. The official results (including not-inconsistent `sat` and `unsat` results, as well as ranks) can be seen in the top rows of Table 1.

Furthermore, we accepted some unsound results due to a variable naming bug, and a buggy loop unrolling pass. This led to some tasks where our verdict differs from the rest of the competition's participants, and therefore, these tasks were deemed inconsistent, and removed from the competition. We hope that this is possible to fix retroactively, because now we know that our results were indeed wrong.

Because the fix was easy once we figured out these problems, we have since published a patch of THETA[2] that corrects this error. We would like to discuss the strength and weaknesses of the fixed version in the rest of this section, because we believe that it reflects the state of THETA's CHC solving capabilities much better than the official results.

To this end, we aim to answer the following research questions:

---

[2]https://github.com/ftsrg/theta/releases/tag/v6.15.3

		LIA	LIA-Lin	LIA-Arrays	LIA-Lin-Arrays	LRA-Lin	BV
comp	sat	52	114	400	45	73	49
	unsat	140	376	8	4	18	123
	rank	5	8	5	4	3	2
fixed	sat	48	585	440	**63**	76	42
	unsat	136	402	12	18	16	126
	rank	5	5	4	1	3	2

Table 1: Results of THETA at the competition (comp) and with the configuration fix (fixed).

	LIA	LIA-Lin	LIA-Arrays	LIA-Lin-Arrays	LRA-Lin	BV
BMC		450		26	88	17
kIND		35		30	1	186
IMC		235		22		
GSAT		4				
BOOL	93	21	395	25	9	13
CART	35	163	23	25	1	1
EXPL	58	91	8	22		3
SVCOMP				26		

Table 2: Solved tasks per configuration per category.

RQ1 How does the fixed version of our CHC-COMP configuration behave on the benchmarks of CHC-COMP'25 in terms of *soundness* and *performance*?

RQ2 Are all configurations in the portfolio capable of solving some unique tasks?

RQ3 Is the performance of he portfolio better than any single configuration, and close to a theoretically optimal solution?

To answer RQ1, we used the same benchmarks as the official competition[3] to run the experiments on our fixed tool again. The results can be seen in the bottom rows of Table 1, alongside a hypothetical rank THETA would have achieved, had we submitted this fixed version of the tool. There were no results that contradict the published verdicts of the benchmark set, other than some, where the published verdict corresponds to the result produced by the submitted version of THETA. We believe these are erroneous, and have flagged this for the competition organizers.

In order to answer RQ2, we counted the number of tasks a configuration solved in a category (keeping in mind, that a prior successful configuration disables a later configuration, so we naturally expect diminished results the later a configuration is). These results are shown in Table 2. Note that these include the inconsistent tasks as well, while Table 1 does not.

Furthermore, for the linear category – them being our focus – we measured each configuration on its own, and also calculated a "virtual best" configuration (taking the best performing configuration for each successfully solved task, therefore representing the result of a theoretical optimal algorithm-selection) to answer RQ3. These results are shown in Figure 4.

4.1 Analysis of the Results

From Table 1, we can see there are some categories where the fixed version underperforms the competition configuration (namely: `LIA` for both, `LRA-Lin` for `unsat`, and `BV` for `sat` results). These do not change the ranking.

---

[3]https://github.com/chc-comp/chc-comp25-benchmarks/commit/ceec2f7740478cc9f114aa87fc54fc167738acdf

(a) LIA-Lin
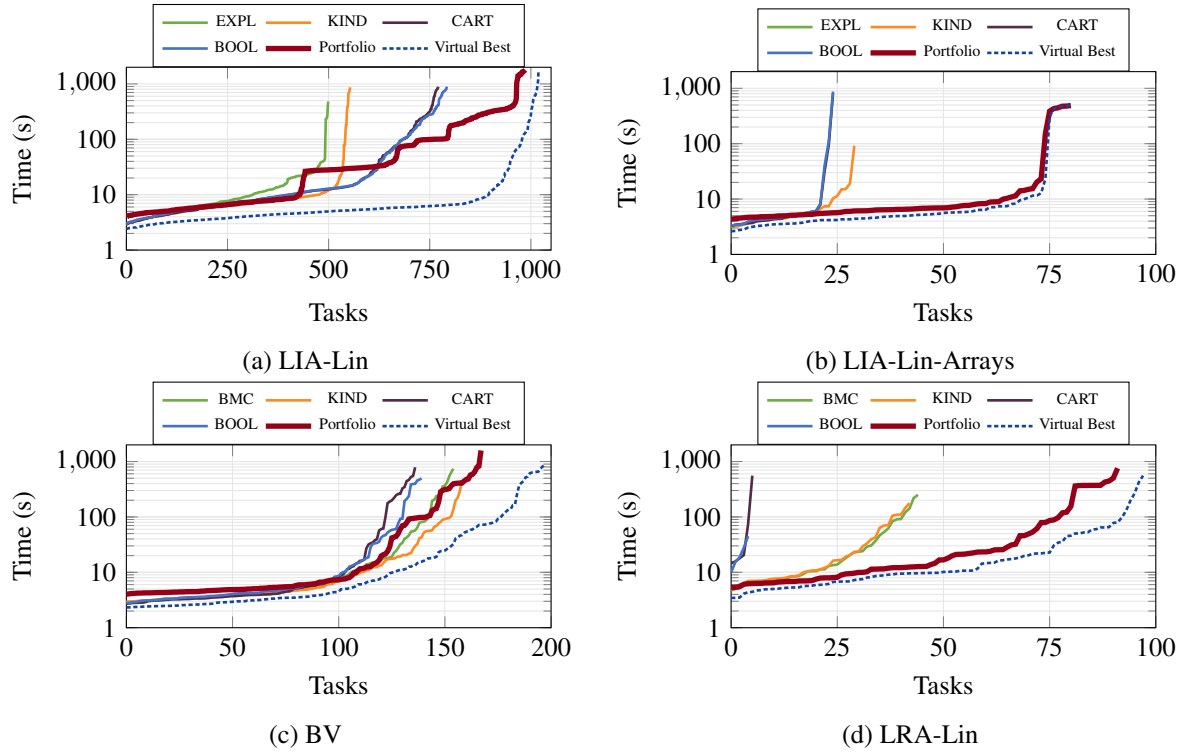
(b) LIA-Lin-Arrays

(c) BV

(d) LRA-Lin

Figure 4: Quantile plots of configurations in the linear categories

We believe these are due to the variable naming and unrolling fixes, and previously, we just happened to find the correct result, but by an erroneous reasoning process.

However, for the majority of categories and results, our fixed solution outperforms the submitted version greatly. While in most cases this does not affect the rank, in 2 categories, it improves it:

1. `LIA-Arrays`: from 5th place to 4th,

2. `LIA-Lin-Arrays`: from 4th place to 1st.

Additionally, the bold **63** number in the `LIA-Lin-Arrays` category's `sat` row shows the highest number of solved tasks among the competition participants. Therefore (at least on the CHC-COMP'25 benchmark set), our tool is the most successful at proving satisfiability of a CHC system with arrays, and only linear clauses.

RQ1: There are no unsound results (when compared to the other participants when they are in agreement), and in two categories, our ranking would improve.

Table 2 shows the number of solved tasks per configuration, per category. There are some categories, where a configuration produced no results, but these may be caused by them being ordered at the end of the sequential portfolio, and thus, an earlier successful configuration would take their chance away from solving the task at hand. However, throughout all categories, every single configuration we feature in our portfolio was successful in at least some of the task. Furthermore, based on further experiments that resulted in Figure 4, we know that no single configurations is completely *shadowed* by another, as there were tasks that were uniquely solved by a single configuration.

RQ2: All configurations in the portfolio solved some tasks in some categories.

In Figure 4, we show the portfolio results in **bold red** and the "virtual best" configuration (taking the quickest correct result from any of the single configuratoins) in dotted blue. We also show the best 4 configurations in each category.

In most cases, the portfolio follows the virtual best selection relatively closely. In Figure 4b, it is almost the same curve, in Figure 4a and Figure 4d the portfolio is slower but solves almost the same number of tasks, but in Figure 4c, the virtual best is much better than the portfolio. Also, k-induction solves almost the same number of tasks as the portfolio, and even quicker in most cases. In the other categories, no single configuration outperforms the portfolio at any point in the runtime meaningfully.

RQ3: In most categories (with linear clauses), the portfolio closely follows the virtual best selection and outperforms all single configurations, but with bitvector arithmetic, the portfolio is far from the virtual best, and k-induction is often better.

These results indicate that the current portfolio strategy is suboptimal for the BV category, and that further tuning or diversification of configurations is necessary to improve performance

4.2 Threats to Validity

The following factors may influence the validity of our experiments.

Internal validity. We used BenchExec [9] to ensure accuracy. We ran our experiments on virtual machines in the cloud computing platform of our university. External factors such as loads on other virtual machines of the host and shared resources may have influenced the results.

External validity. The CHC-COMP benchmark suite is considered the standard for academic benchmarking of CHC solving algorithms and tools. Still, evaluation results might not generalize well to other sources of problems.

Construct validity. The metrics of the evaluation were carefully chosen to accurately describe the performance of our algorithm, while not using misleading statistics such as memory usage, given our tool is running in a managed environment (JVM). We concentrate on statistics that meaningfully impact a potential user of our tool: the number of solved tasks, and the time it takes to produce a solution.

5 Tool Setup and Configuration

THETA is a modular and highly configurable framework [18], with support for multiple input languages, algorithms, and solvers. For CHC-based verification tasks, we recommend using the following invocation: `./chc <input>`. This runs our portfolio-based approach, which chooses a suitable sequence of configurations based on the input file automatically.

6 Software Project and Data Availability

THETA is developed and maintained by the Critical Systems Research Group at the Budapest University of Technology and Economics. The framework is available open-source on GitHub[4] under the Apache 2.0 license. The version used for the experiments in this paper is available at [4].

---

[4] https://github.com/ftsrg/theta

References

[1] Asadi, S., Blicha, M., Hyvärinen, A., Fedyukovich, G., Sharygina, N.: Farkas-Based Tree Interpolation. In: Pichardie, D., Sighireanu, M. (eds.) Static Analysis. pp. 357–379. Springer International Publishing, Cham (2020). doi:10.1007/978-3-030-65474-0_16

[2] Baier, D., Beyer, D., Friedberger, K.: JavaSMT3: Interacting with SMT Solvers in Java. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification. pp. 195–208. Springer International Publishing, Cham (2021). doi:10.1007/978-3-030-81688-9_9

[3] Bajczi, L., Molnár, V.: Solving Constrained Horn Clauses as C Programs with CHC2C. In: Model Checking Software: 30th International Symposium, SPIN 2024, Luxembourg City, Luxembourg, April 8–9, 2024, Proceedings. p. 146–163. Springer-Verlag, Berlin, Heidelberg (2024). doi:10.1007/978-3-031-66149-5_8

[4] Bajczi, L., Mondok, M., Molnár, V.: Thetachc - verifier archive (May 2025). doi:10.5281/zenodo.15537903

[5] Bajczi, L., Szekeres, D., Mondok, M., Ádám, Z., Somorjai, M., Telbisz, C., Dobos-Kovács, M., Molnár, V.: EmergenTheta: Verification Beyond Abstraction Refinement (Competition Contribution). In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part III. Lecture Notes in Computer Science, vol. 14572, pp. 371–375. Springer (2024). doi:10.1007/978-3-031-57256-2_23

[6] Ball, T., Podelski, A., Rajamani, S.K.: Boolean and Cartesian abstraction for model checking C programs. International Journal on Software Tools for Technology Transfer **5**, 49–58 (2003). doi:10.1007/s10009-002-0095-0

[7] Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., et al.: cvc5: A versatile and industrial-strength SMT solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 415–442. Springer (2022). doi:10.1007/978-3-030-99524-9_24

[8] Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker blast: Applications to software engineering. International Journal on Software Tools for Technology Transfer **9**, 505–525 (2007). doi:10.1007/s10009-007-0044-z

[9] Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. International Journal on Software Tools for Technology Transfer **21**(1), 1–29 (2019). doi:10.1007/s10009-017-0469-y

[10] Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. In: FMICS 2002, ICALP 2002 Satellite Workshop (2002). doi:10.1016/S1571-0661(04)80410-9

[11] Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: TACAS (1999). doi:10.1007/3-540-49059-0_14

[12] Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. Journal of the ACM (JACM) **50**(5), 752–794 (2003). doi:10.1145/876638.876643

[13] De Angelis, E., Vediramana Krishnan, H.G.: Competition of Solvers for Constrained Horn Clauses (CHC-COMP 2023). In: International TOOLympics Challenge, pp. 38–51. Springer (2024). doi:10.1007/978-3-031-67695-6_2

[14] De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008). doi:10.1007/978-3-540-78800-3_24

[15] Dobos-Kovács, M., Vörös, A.: Evaluation of SMT solvers in abstraction-based software model checking. In: Proceedings of the 11th Latin-American Symposium on Dependable Computing. pp. 109–116 (2022). doi:10.1145/3569902.3570187

[16] Eén, N., Mishchenko, A., Brayton, R.: Efficient implementation of property directed reachability. In: FM-CAD (2011). doi:10.5555/2157654.2157675

[17] Gurfinkel, A.: Program verification with constrained horn clauses. In: International Conference on Computer Aided Verification. pp. 19–29. Springer (2022). doi:10.1007/978-3-031-13185-1_2

[18] Hajdu, Á., Micskei, Z.: Efficient Strategies for CEGAR-based Model Checking. Journal of Automated Reasoning **64**(6), 1051–1091 (2020). doi:10.1007/s10817-019-09535-x

[19] Lange, T., Neuhäußer, M.R., Noll, T., Katoen, J.P.: Ic3 software model checking. International Journal on Software Tools for Technology Transfer **22**(2), 135–161 (2020). doi:10.1007/s10009-019-00547-x

[20] McMillan, K.L.: Interpolation and SAT-Based Model Checking. In: CAV (2003). doi:10.1007/978-3-540-45069-6_1

[21] McMillan, K.L.: Applications of Craig interpolants in model checking. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 1–12. Springer (2005). doi:10.1007/978-3-540-31980-1_1

[22] Molnár, V.: Extensions and Generalization of the Saturation Algorithm in Model Checking. Phd thesis, Budapest University of Technology and Economics (2019)

[23] Mondok, M., Molnár, V.: Efficient Manipulation of Logical Formulas as Decision Diagrams. In: 31st PhD Mini-Symposium (2024). doi:10.3311/MINISY2024-012

[24] Otoni, R., Marescotti, M., Alt, L., Eugster, P., Hyvärinen, A., Sharygina, N.: A solicitous approach to smart contract verification. ACM Transactions on Privacy and Security **26**(2), 1–28 (2023)

[25] Sheeran, M., Singh, S., Stålmarck, G.: Checking Safety Properties Using Induction and a SAT-Solver. In: FMCAD (2000). doi:10.1007/3-540-40922-X_8

[26] Somorjai, M., Dobos-Kovács, M., Ádám, Z., Bajczi, L., Vörös, A.: Bottoms Up for CHCs: Novel Transformation of Linear Constrained Horn Clauses to Software Verification. Electronic Proceedings in Theoretical Computer Science **402**, 105–117 (Apr 2024). doi:10.4204/eptcs.402.11

[27] Telbisz, C., Bajczi, L., Szekeres, D., Vörös, A.: Theta: Various approaches for concurrent program verification (competition contribution). In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 260–265. Springer (2025)

[28] Tonetta, S.: Abstract Model Checking without Computing the Abstraction. In: Formal Methods (2009). doi:10.1007/978-3-642-05089-3_7

[29] Tóth, T., Hajdu, Á., Vörös, A., Micskei, Z., Majzik, I.: Theta: a Framework for Abstraction Refinement-Based Model Checking. In: Stewart, D., Weissenbacher, G. (eds.) Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design. pp. 176–179 (2017). doi:10.23919/FMCAD.2017.8102257