EPTCS 434

Proceedings of the

12th Workshop on Horn Clauses for Verification and Synthesis

Zagreb, Croatia, 22 July 2025

Edited by: Emanuele De Angelis and Florian Frohn

Published: 30th October 2025
DOI: 10.4204/EPTCS.434
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Preface

This volume contains the post-proceedings of the 12th Workshop on Horn Clauses for Verification and Synthesis (HCVS 2025, `https://www.sci.unich.it/hcvs25/`), which took place in Zagreb, Croatia on July 22, 2025, as affiliated workshop of the 37th International Conference on Computer Aided Verification (CAV 2025).

The aim of the HCVS workshop series is to bring together researchers working in the communities of Constraint/Logic Programming (e.g., ICLP and CP), Program Verification (e.g., CAV, TACAS, and VMCAI), and Automated Deduction (e.g., CADE), on the topic of Horn clause based analysis, verification and synthesis. Horn clauses for verification and synthesis have been advocated by these communities at different times and from different perspectives, and HCVS is organized to stimulate interaction and a fruitful exchange and integration of experiences.

HCVS 2025 follows previous meetings: HCVS 2024 in Luxembourg (ETAPS 2024), HCVS 2023 in Paris (ETAPS 2023), HCVS 2022 in Munich (ETAPS 2022), HCVS 2021 in Luxembourg (online, ETAPS 2021), HCVS 2020 in Dublin, Ireland (ETAPS 2020), HCVS 2019 in Prague, Czech Republic (ETAPS 2019), HCVS 2018 in Oxford, UK (CAV, ICLP and IJCAR at FLoC 2018), HCVS 2017 in Gothenburg, Sweden (CADE 2017), HCVS 2016 in Eindhoven, The Netherlands (ETAPS 2016), HCVS 2015 in San Francisco, CA, USA (CAV 2015), and HCVS 2014 in Vienna, Austria (VSL).

The workshop program featured six presentations of four original papers and two presentation-only papers, two invited talks by Grigory Fedyukovich (Florida State University, USA) and Gennaro Parlato (University of Molise, Italy), and a report on the CHC competition (CHC-COMP).

We wish to express our gratitude to the authors who submitted papers, the speakers, the invited speakers, and the program committee members for the work they have generously done. We would like to thank Fabio Fioravanti (University of Chieti-Pescara, Italy) of the HCVS steering committee, and Grigory Fedyukovich (Florida State University, USA), chair of the CAV 2025 workshops, for their help in making the event possible. Finally, we warmly thank the EasyChair organization for supporting the various tasks related to the selection of the contributions, and EPTCS for hosting the proceedings.

October 2025,
Emanuele De Angelis and Florian Frohn

Program Chairs

Emanuele De Angelis, CNR-IASI, Italy
Florian Frohn, RWTH Aachen, Germany

Program Committee

Nikolaj Bjørner, Microsoft, USA
Martin Blicha, University of Lugano, Switzerland
Konstantin Britikov, University of Lugano, Switzerland
Catherine Dubois, ENSIIE-Samovar, France
Gidon Ernst, Ludwig Maximilian University of Munich, Germany
Zafer Esen, Uppsala University, Sweden
Grigory Fedyukovich, Florida State University, USA
Carsten Fuhs, Birkbeck, University of London, UK

Building Automated Reasoning Tools using the FreqHorn Framework

Grigory Fedyukovich
Florida State University, USA

Originally developed in 2017, the FreqHorn framework follows the Syntax-Guided Synthesis paradigm to automatically synthesize solutions for systems of Constrained Horn Clauses (CHC). Its distinguishing feature was the automatic construction of formal grammars for each uninterpreted predicate based on either its syntax, data, or bounded proofs based on Craig interpolation. Grammars are useful for the systematic enumeration of solution candidates and checking them with an SMT solver. Recently, the framework was extended to reason about CHCs with arrays, algebraic data types, and multi-phase systems.

In this talk, I will give an overview of various features of FreqHorn that enable handling complex CHC systems. Moreover, I will demonstrate how its infrastructure has already been used to build CHC-based automated reasoning tools applicable for answering questions on program termination, program equivalence, precondition generation, and automated test case generation. Furthermore, FreqHorn is frequently used in teaching graduate-level courses on logic and program verification, in which students find it easy to extend and maintain.

Unifying Automata Theory and Constrained Horn Clauses for Scalable Program Verification

Gennaro Parlato
University of Molise, Italy

In this talk, I present a body of work developed in collaboration with Marco Faella, exploring the intersection of automata theory and Constrained Horn Clauses (CHCs) to advance program verification and synthesis.

We introduce Symbolic Data Tree Automata (SDTAs) [FP22], a novel automata class generalizing classical tree automata through symbolic reasoning over structured data domains. SDTAs seamlessly integrate finite-state automata techniques with logical reasoning offered by CHCs, enabling robust and flexible analyses of computations on hierarchical or tree-shaped structures. While SDTA emptiness checking is generally undecidable, we demonstrate a practical reduction to CHC satisfiability, leveraging existing, efficient CHC solvers.

To extend logical expressiveness, we propose MSO-D (Monadic Second-Order logic with Data) [FP22], an extension of standard MSO on trees with predicates from an underlying data theory, designed for expressive reasoning about complex data structures. Although MSO-D is Turing-powerful in general, we identify a significant fragment whose satisfiability reduces directly to SDTA emptiness, thus facilitating automated verification using CHC solvers. This fragment retains considerable expressiveness, enabling characterization of diverse data structures commonly encountered in verification problems, as well as solving certain classes of infinite-state games [FP23].

For linear-time specifications, we introduce an automata-theoretic approach for verifying properties expressed in $\mathrm{LTL}_f^{\mathrm{MT}}$ (Linear Temporal Logic over finite traces Modulo Theories) [FP24]. This logic augments LTL_f with quantifier-free constraints from rich data theories. Our approach translates $\mathrm{LTL}_f^{\mathrm{MT}}$ into Symbolic Data Word Automata (SDWAs), where transitions are governed by theory constraints. Though satisfiability for both $\mathrm{LTL}_f^{\mathrm{MT}}$ and SDWAs is undecidable, our reduction to CHC satisfiability proves highly effective for model checking and runtime monitoring, as demonstrated by competitive experimental results against specialized tools.

Finally, I present our latest work [FP25] on verifying programs that manipulate dynamic, tree data structures–typically a challenging task requiring intricate manual proofs. We introduce a unified framework based on knitted-tree encodings, which represent program executions as structured trees capturing inputs, outputs, and intermediate states. This compositional representation supports modular invariants and enables fully automated verification (e.g., memory safety) via CHC solvers. Our results demonstrate that this approach substantially simplifies and scales reasoning about complex, heap-manipulating programs.

References

[FP22] M. Faella & G. Parlato (2022): *Reasoning About Data Trees Using CHCs.* In S. Shoham & Y. Vizel, editors: *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-*

10, 2022, Proceedings, Part II, *Lecture Notes in Computer Science* 13372, Springer, pp. 249–271, doi:`10.1007/978-3-031-13188-2_13`.

[FP23] M. Faella & G. Parlato (2023): *Reachability Games Modulo Theories with a Bounded Safety Player*. In B. Williams, Y. Chen & J. Neville, editors: *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, AAAI Press, pp. 6330–6337, doi:`10.1609/AAAI.V37I5.25779`.

[FP24] M. Faella & G. Parlato (2024): *A Unified Automata-Theoretic Approach to LTL$_f$ Modulo Theories*. In U. Endriss, F.S. Melo, K. Bach, A.J.B. Diz, J.M. Alonso-Moral, S. Barro & F. Heintz, editors: *ECAI 2024 - 27th European Conference on Artificial Intelligence, 19-24 October 2024, Santiago de Compostela, Spain - Including 13th Conference on Prestigious Applications of Intelligent Systems (PAIS 2024)*, *Frontiers in Artificial Intelligence and Applications* 392, IOS Press, pp. 1254–1261, doi:`10.3233/FAIA240622`.

[FP25] M. Faella & G. Parlato (2025): *Verifying Tree-Manipulating Programs via CHCs*. In R. Piskac & Z. Rakamaric, editors: *Computer Aided Verification - 37th International Conference, CAV 2025, Zagreb, Croatia, July 23-25, 2025, Proceedings, Part I*, *Lecture Notes in Computer Science* 15931, Springer, pp. 3–28, doi:`10.1007/978-3-031-98668-0_1`.

Finding Regular Herbrand Models for CHCs using Answer Set Programming

Grégoire Maire
ENS Rennes
gregoire.maire@ens-rennes.fr

Thomas Genet
Univ Rennes, IRISA, Inria
genet@irisa.fr

We are interested in proving satisfiability of Constrained Horn Clauses (CHCs) over Algebraic Data Types (ADTs). We propose to prove satisfiability by building a tree automaton recognizing the Herbrand model of the CHCs. If such an automaton exists then the model is said to be *regular*, i.e., the Herbrand model is a regular set of atoms. Kostyukov & al. [5] have shown how to derive an automaton when CVC4 finds a finite model of the CHCs. We propose an alternative way to build the automaton using an encoding into a SAT problem using Clingo, an Answer Set Programming (ASP) tool. We implemented a translation of CHCs with ADTs into an ASP problem. Combined with Clingo, we obtain a semi-complete satisfiability checker: it finds a tree automaton if a regular Herbrand model exists or finds a counter-example if the problem is unsatisfiable.

We are interested in the automatic verification of programs manipulating Algebraic Data Types (ADTs). The analysis of such programs is challenging as soon as the ADTs are recursive because they define unbounded data structures. When ADTs are recursive, tree automata [1] provide an efficient way to finitely represent unbounded sets of such ADTs. In [8, 3, 4, 5], verifying a property ϕ on a program P consists in building a tree automaton recognizing a set of all the computations of the program and in checking that the property is true on this set. When P is represented by a set of functions (resp. a term rewriting system), the property ϕ is expressed as a set of results that should not be reachable when applying the semantics of P on initial function calls (resp. rewriting initial terms with P). In this setting the tree automaton finitely represents the set of all values (resp. terms) that are reachable when applying the semantics of P (resp. rewriting with P). In the context of program verification using Constrained Horn Clauses (CHC for short), this is adapted as follows: P is represented by a set of Horn clauses and the property ϕ is a negative formula, i.e., $\phi \overset{def}{=} (\psi \Rightarrow \bot)$. To prove that $P \Rightarrow (\psi \Rightarrow \bot)$ is valid, we build a tree automaton finitely representing the least Herbrand model M of P and we check that the formula ψ does not hold in M. This entails that ψ is *not* a logical consequence of P. Thus, $P \Rightarrow (\psi \Rightarrow \bot)$ is valid. In the following, if a model can be represented by a tree automaton we call it a *regular model*, i.e., the Herbrand model is a regular set of atoms.

1 An introductory example

For instance, let $nat = z \mid s(nat)$ be the ADT defining natural numbers. Let P be the set of CHCs defining $even(x)$ and $odd(x)$ as the usual predicates over numbers and $plus(x,y,z)$ as the predicate such that $z = x + y$. Let $\phi_1 \overset{def}{=} \forall x\, y\, z.\ even(x) \wedge even(y) \wedge plus(x,y,z) \wedge odd(z) \Rightarrow \bot$ be the (negative) property we want to prove on P. Since the ADT of natural numbers is recursive, Herbrand models of P are unbounded. However, we can finitely represent such an unbounded model using a tree automaton:

$$\begin{array}{lll}
z \rightarrow \#2 & odd(\#1) \rightarrow \#0 & plus(\#1,\#2,\#1) \rightarrow \#0 \\
s(\#2) \rightarrow \#1 & even(\#2) \rightarrow \#0 & plus(\#1,\#1,\#2) \rightarrow \#0 \\
s(\#1) \rightarrow \#2 & plus(\#2,\#1,\#1) \rightarrow \#0 & plus(\#2,\#2,\#2) \rightarrow \#0
\end{array}$$

In this automaton, #0, #1, #2 are the states of the automaton. A term is recognized by a state if it can be rewritten to this state using the transitions. For instance, the state #2 recognizes the term z, the state #1 recognizes $s(z) \to s(\#2) \to \#1$, i.e., #2 recognizes even numbers and #1 recognizes odd numbers. Finally, the state #0 recognizes all the atoms that are true in the considered Herbrand model, e.g, $even(s(s(z)))$, $odd(s(z))$. Note that the model recognized by this automaton is not the least Herbrand model but an over-approximation. In particular, this automaton recognizes $plus(s(z),s(z),s(s(z)))$ which is part of the least Herbrand model but also $plus(s(z),s(z),z)$ which is not. On this model, the property ϕ_1 is true. In particular, with the rule $plus(\#2,\#2,\#2) \to \#0$ we can see that summing two even numbers always result into an even number. Since the negative property ϕ_1 is true on an over-approximation of the least Herbrand model then it is also true on the least Herbrand model.

To infer such an automaton, the tool RInGen by Kostyukov & al. [5] use the finite model finder of CVC4. They transform the input problem $(P \wedge \phi)$ over the theory of ADTs into a problem in the Equality Logic with Uninterpreted Functions (EUF). Then, if there exists a finite model of the problem in EUF, they show how to derive a tree automaton recognizing an Herbrand model in the ADT theory satisfying $P \wedge \phi$ and proving $P \Rightarrow \phi$.

2 Building automata recognizing regular models using ASP

In this paper, we report preliminary experiments on an alternative way to build such an automaton by an encoding into a SAT problem using Clingo [2], an Answer Set Programming (ASP) tool. Complex automata inference with Clingo has already been experimented in [6, 7]. Given a set of Prolog-style clauses, Clingo searches for a Herbrand model of this set of clauses. Unlike usual Prolog interpreter, Clingo is guaranteed to terminate and outputs the Herbrand models as soon as the models are finite. However, how discussed above, the Herbrand models we look for are *not* finite but can be *finitely* represented using tree automata. Here is a possible encoding of P and ϕ_1 in Clingo. We first set the maximal number of states in the automaton we search for.

```
#const maxState=2.
state(1..maxState). % this shortcut builds facts state(1). and state(2).
```

The following lines define the tree automaton rules for the abstraction of the ADT. We encode a rule of the form $s(\#1) \to \#2$ by the fact `rule(s(1),2)` The automaton we want to build for terms of the ADT is expected to be complete (any term should be recognized by *at least* one state) and deterministic (any term should be recognized by *at most* one state). This is easily encoded using Clingo's cardinality constraints.

```
1 {rule(z, Q): state(Q)} 1.
1 {rule(s(Q0), Q): state(Q)} 1 :-state(Q0).
```

In the first line above, the brackets around the fact `rule(z,Q)` mean that this fact may or may not appear in the searched model. By adding 1 on the left, we impose that *at least* one fact of this kind appears in the model. By adding 1 on the right we impose that *at most* one fact of this form appears in the model. The annotation `state(Q)` forces `Q` to be one of the states. Thus, if a model is found it will necessarily have exactly one fact `rule(z,1)` or `rule(z,2)` (since we have here only 2 states). The second line ensures there is exactly one state Q such that $s(Q_0) \to Q$ for all states Q_0. The following lines essentially give the types and cardinality of the relations *even*, *odd* and *plus*. We provide those information but we want to infer the relation themselves. Again, because of the brackets, these lines only say that those facts may or may not appear in the model.

```
{even(Q0)} :-state(Q0).
{odd(Q0)} :-state(Q0).
{plus(Q0, Q1, Q2)} :-state(Q0), state(Q1), state(Q2).
```

Finally, we can state the CHCs of our satisfiability problem. They are directly translated into Clingo clauses where terms are replaced by the corresponding states and transitions. For instance, one clause defining the *even* predicate is $even(s(X)) :- odd(X)$. In the encoding, since predicates ranges over states and not terms, we cannot directly represent an atom over the term $s(X)$. Instead, we encode this using several facts, i.e., a state Q_1 and a rule $s(Q_0) \to Q_1$. This results into the following set of Clingo clauses.

```
% Translation of : even(z).
even(Q0) :-rule(z, Q0).
% Translation of : even(s(X)) :-odd(X).
even(Q2) :-odd(Q1), rule(s(Q1), Q2).
% Translation of : odd(s(X)) :-even(X).
odd(Q2) :-even(Q1), rule(s(Q1), Q2).
% Translation of : plus(z, X, X).
plus(Q1, Q0, Q0) :-rule(z, Q1), state(Q0).
% Translation of : plus(s(X), Y, s(Z)) :-plus(X, Y, Z).
plus(Q6, Q3, Q7) :-plus(Q1, Q3, Q5), rule(s(Q1), Q6), rule(s(Q5), Q7).
% Translation of : :-even(X), even(Y), plus(X, Y, Z), odd(Z).
:-even(Q0), even(Q1), plus(Q0, Q1, Q2), odd(Q2).
```

We prototyped this translation and the satisfiability checking in OCaml and Clingo: `https://gitlab.inria.fr/regular-pv/regularmodels`. The translation is very close to the one above except that it also uses a predicate `stateType(Q,t)` to distinguish states w.r.t. the type `t` of the terms they recognize. Another difference is that bodies of initial CHCs may contain equalities $X = Y$ or disequalities $X != Y$ ranging over terms. Equalities can be encoded by equalities on states because the automaton is deterministic: if terms are equal then so are the states. However, note that different terms may be recognized by the same state. Hence, the body of the clause may be true on states though it is not on the recognized terms. This results into an over-approximation of the Herbrand model which is safe w.r.t. the property that is a negative clause. On the opposite, encoding term disequalities by state disequalities is not safe: a disequality $X != Y$ in the body may be satisfied by two different terms t_1 and t_2 though they are recognized by the same state. This would result into an under-approximation of the Herbrand model which is not safe. As a result we define the `diffApprox(Q1,Q2)` predicate over-approximating the $!=$ relation on terms. This predicate is true if $Q1$ and $Q2$ recognizes at least two different terms.

Finally, our satisfiability procedure generates Clingo specifications with increasing values of `maxStates` until one solution is found. For each value of `maxStates`, we generate two specifications: one for satisfiability checking and another (with small modifications) to search for a counterexample. Note that, given a value of `maxStates`, if Clingo fails to find a model (and if Clingo is complete) then we have a guarantee that there exists no regular Herbrand model that can be recognized by an automaton of `maxStates` states. Here is the output of our prototype on the example of Section 1.

```
Searching for a counterexample with 1 state
Searching for a model with 1 state
Searching for a counterexample with 2 states
Searching for a model with 2 states
ADT Transitions:        Predicates:
Z -> 2                  odd(1)              plus(1,2,1)
S(2) -> 1               even(2)             plus(1,1,2)
S(1) -> 2               plus(2,1,1)         plus(2,2,2)

Success! Clauses are satisfiable by a Herbrand model recognized by a tree
automaton with 2 states
```

Note that in the tree automaton of Section 1, we also generated a state (#0) and transitions (e.g. $plus(\#1,\#2,\#1) \rightarrow \#0$) to recognize the terms rooted by predicate symbols. However those transitions are useless for verification of CHCs and are, thus, discarded in the output of our prototype. By iteratively increasing `maxStates`, we have a semi-complete tool to check for satisfiability of CHCs with ADTs: if there exists a regular Herbrand model we will find it. This was also the case with RInGen [5] where semi-completeness relies on completeness of CVC4 finite model-finder.

3 Experimental evaluation

With regards to efficiency, our prototype is not yet as efficient as RInGen. This is essentially due to the fact that our Clingo encoding is too general: each Clingo specification may have several equivalent solutions, i.e., several equivalent Herbrand models. Since efficiency of the Clingo solving highly depends on the number of possible solutions, we need to reduce the number of solutions to improve the efficiency of our prototype. For instance, with the Clingo specification of the previous section, encoding `plus(X,Y,Z)`, `even(X)`, and `odd(X)`, there are two equivalent solutions. The solution automaton presented in the above section recognizes odd numbers in state `1` and even numbers in state `2`. However, the generated Clingo specification has a second equivalent and symmetrical solution where odd numbers are recognized in state `2` and even numbers in state `1`.

We studied the impact symmetries on Clingo's solving efficiency using a more complex verification problem using two ADTs: the type *elt* of elements and the type *list* of lists of *elt*. The ADT *elt* contains a finite set of k constants where $k \in \mathbb{N}$, i.e., $elt = a_1|\ldots|a_k$. The *list* ADT is defined by $list = nil \mid cons(elt, list)$. Let P be the set of CHCs defining $member(x,l)$ as the predicate which is true if the element x belongs to the list l, $notMember(x,l)$ as the negation of $member(x,l)$, and $rev(l_1,l_2)$ such that l_2 is l_1 reversed. Assume that we want to prove the property that an element belongs to a list if and only if it belongs to the reverse of this list. This property can be encoded by the following two negative formulas $phi_2 \stackrel{def}{=} \forall x\ l_1\ l_2.\ member(x,l_1) \wedge reverse(l_1,l_2) \wedge notMember(x,l_2) \Rightarrow \bot$ and $phi_3 \stackrel{def}{=} \forall x\ l_1\ l_2.\ notMember(x,l_1) \wedge reverse(l_1,l_2) \wedge member(x,l_2) \Rightarrow \bot$.

Having an algebraic data-type *elt* whose size k vary makes it possible to increase the complexity of the verification problem by increasing k. We tried to prove the above verification problem ($P \Rightarrow \phi_2 \wedge \phi_3$) for values of k ranging from 2 to 4. We experimented with RInGen and our prototype. For $k = 2$, RInGen solves it in 0.075s while our tool solves it in 0.395s. For $k = 3$, RInGen solves it in 1.211s while our tool solves it in 2700s (45 minutes!). This example shows that a naive ASP-encoding will fail to efficiently build regular models. The influence of symmetries can be observed by asking Clingo to generate the number of solutions for a given input specification. With $k = 2$ the number of solutions is greater than 700 millions. With $k = 3$ the number of solutions is so huge that Clingo fails to output it.

We modified by hand the Clingo specifications generated by our tool in order to apply some simple symmetry breaking techniques. The objective is to find an order on states that is restrictive enough to discard equivalent solutions and permissive enough not to loose any valid solution. We applied this to the above verification problem for values of k from 2 to 4. We sum-up all those experiments in the table Figure 1, where we compare the execution time for RInGen (using CVC4 as a backend), the execution time for our ASP-prototype, the number of equivalent models for our ASP-prototype, the execution time for our ASP-prototype with symmetry breaking modification done by hand, and finally the corresponding number of models with symmetry breaking.

$k = \|elt\|$	RInGen CVC4 (sec.)	ASP-prot. (sec.)	ASP-prot. # models	ASP-prot. sym. break. (sec.)	ASP-prot. sym. break. # models
2	0.075	0.395	700 M+	0.035	12
3	1.211	2700	Timeout	0.616	Timeout
4	Timeout	Timeout	Timeout	Timeout	Timeout

Figure 1: Experiments with RInGen, our prototype and our prototype with symmetry breaking

In this table, we can remark on line $k = 2$ that even a simple symmetry breaking dramatically reduce the number of considered models. The effect on efficiency is valuable for $k = 2$ but is really significant for $k = 3$ where the computation time decreases from 2700s to 0.616s. We even get an execution time that is lower than the one of RInGen. However, our symmetry breaking can still be improved since the number of models for $k = 3$ is still too big to be outputted by Clingo. Finally, the last remark is that no implementation can solve this problem for $k = 4$ and, thus, there is still room for improvements!

We believe that the basic symmetry breaking we carried out by hand is correct, i.e., that it does not jeopardize the semi-completeness of the approach. However, this has to be proven. If correct, our symmetry breaking strategy has to be integrated in our prototype. The proof and implementation are left for future work. Besides, we believe that using an even more aggressive symmetry breaking technique could yield a regular model finder more efficient than RInGen because Clingo's solving core is pure SAT-solving. This has to be investigated further. Another way to improve efficiency is to use a modular solving based on the Regular Language Typing approach of [4]. Finally, moving automata inference from CVC4 to ASP-based solvers should open ways to infer automata with constraints, e.g., tree automata with arithmetic constraints to verify programs with ADTs containing numerical values.

Acknowledgments Many thanks to Théo Losekoot and Jacques Nicolas for fruitful discussions about Clingo.

References

[1] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, C. Löding, S. Tison & M. Tommasi (2008): *Tree Automata Techniques and Applications*. Available at `https://inria.hal.science/hal-03367725`.

[2] M. Gebser, R. Kaminski, B. Kaufmann & T. Schaub (2019): *Multi-shot ASP solving with clingo*. *TPLP* 19(1), pp. 27–82, doi:10.1017/S1471068418000054.

[3] T. Genet, T. Haudebourg & T. Jensen (2018): *Verifying Higher-Order Functions with Tree Automata*. In: *FoSSaCS'18*, *LNCS* 10803, Springer, doi:10.1007/978-3-319-89366-2_31.

[4] T. Haudebourg, T. Genet & T. Jensen (2020): *Regular Language Type Inference with Term Rewriting*. In: *ICFP'20*, 4, ACM, pp. 112:1–112:29, doi:10.1145/3408994.

[5] Y. Kostyukov, D. Mordvinov & G. Fedyukovich (2021): *Beyond the Elementary Representations of Program Invariants over Algebraic Data Types*. In: *PLDI '21*, ACM, pp. 451–465, doi:10.1145/3453483.3454055.

[6] T. Losekoot, T. Genet & T. Jensen (2023): *Automata-based Verification of Relational Properties of Functions over Data Structures*. In: *FSCD'23*, 260, LIPIcs, doi:10.4230/LIPICS.FSCD.2023.7.

[7] T. Losekoot, T. Genet & T. Jensen (2024): *Verification of Programs with ADTs Using Shallow Horn Clauses*. In: *SAS'2024*, *LNCS* 14995, Springer, pp. 242–267, doi:10.1007/978-3-031-74776-2_10.

[8] Y. Matsumoto, N. Kobayashi & H. Unno (2015): *Automata-Based Abstraction for Automated Verification of Higher-Order Tree-Processing Programs*. In: *APLAS'15*, *LNCS* 9458, Springer, pp. 295–312, doi:10.1007/978-3-319-26529-2_16.

Semantic Properties of Computations Defined by Elementary Inference Systems*

Salvador Lucas
DSIC & VRAIN, Universitat Politècnica de València, Spain
slucas@dsic.upv.es

We consider sets/relations/computations defined by *Elementary Inference Systems* $\mathscr{I}$, which are obtained from Smullyan's *elementary formal systems* using Gentzen's notation for inference rules, and proof trees for atoms $P(t_1,\ldots,t_n)$, where predicate P represents the considered set/relation/computation. A first-order theory $\mathsf{Th}(\mathscr{I})$, actually a set of definite Horn clauses, is given to $\mathscr{I}$. Properties of objects defined by $\mathscr{I}$ are expressed as first-order sentences F, which are proved true or false by *satisfaction* $\mathscr{M} \models F$ of F in a *canonical* model $\mathscr{M}$ of $\mathsf{Th}(\mathscr{I})$. For this reason, we call F a *semantic property* of $\mathscr{I}$. Since canonical models are, in general, incomputable, we show how to (dis)prove semantic properties by satisfiability in an *arbitrary* model $\mathscr{A}$ of $\mathsf{Th}(\mathscr{I})$. We apply these ideas to the analysis of properties of programming languages and systems whose computations can be described by means of an elementary inference system. In particular, rewriting-based systems.

1 Introduction

Elementary formal systems [46] provide an appropriate device for the definition and combination of sets, relations, and hence of computational relations, which is amenable for *mechanization*. The *operational semantics* of computational systems and programming languages is often given by means of a formal system, usually presented as a set of *inference rules* which are used to *prove* goals $P(t_1,\ldots,t_n)$ for some predicate symbol P (representing the considered set of elements or tuples of elements) and terms $t_1,\ldots,t_n$ (representing components or tuples of components). In [41], Plotkin recalls the role of Smullyan's formal systems [46] in the development of his Structural Operational Semantics (SOS [40, 42]),[1] which is widely used in the semantic description of programming languages since the 1980s, see, e.g., [23]. Plotkin also mentions Barendregt's PhD thesis [3] where λ-calculus is described using inference rules, see [3, Appendix I]. In particular, he displays this rule (from [3, page 12]):

$$\frac{N \triangleright N'}{M\,N \triangleright M\,N'} \tag{1}$$

where, as in [21], we use $\triangleright$ instead of Barendregt's original $\geq$ to denote β-reduction. Rule (1) expresses that β-reduction is *propagated* on the *second* argument of λ-calculus *application* (with binary operator $\_\,\_$). There is a similar rule for propagation on the first argument as well.

Despeyroux introduced the term *Natural Semantics* [11] to refer to *the purely 'formal system' part* of SOS which actually relies on Gentzen's Natural Deduction [16, 43], where proofs of computations are represented by means of *proof trees*. In order to *reason* about computations described with such formal systems, the use of *first-order formulas* which can be proved true or false of the defined object is a natural choice to *express* properties [23, Section 1.1, last paragraph]. As posed by Kahn,

*Supported by project PID2021-122830OB-C42 funded by MCIN/AEI/10.13039/501100011033 and by "ERDF A way of making Europe" and by the grant CIPROM/2022/6 funded by Generalitat Valenciana

[1] However, [42] contains no reference to Smullyan.

$$(\mathrm{Rf})\quad \frac{}{x \to^* x} \qquad (\mathrm{Co})\quad \frac{x \to y \quad y \to^* z}{x \to^* z} \qquad (\mathrm{HC})_{(2)}\quad \frac{}{x \geq 0}$$

$$(\mathrm{HC})_{(3)}\quad \frac{x \geq y}{\mathsf{s}(x) \geq \mathsf{s}(y)} \qquad (\mathrm{HC})_{(4)}\quad \frac{x \to^* \mathsf{s}(\mathsf{s}(\mathsf{0}))}{\mathsf{peven}(x)} \qquad (\mathrm{HC})_{(5)}\quad \frac{x \to^* \mathsf{s}(\mathsf{0})}{\mathsf{odd}(x)}$$

$$(\mathrm{HC})_{(6)}\quad \frac{x \to^* \mathsf{0}}{\mathsf{zero}(x)} \qquad (\mathrm{HC})_{(7)}\quad \frac{x \geq \mathsf{s}(\mathsf{0})}{\mathsf{s}(\mathsf{s}(x)) \to x}$$

Figure 1: Elementary inference system $\mathcal{I}(\mathcal{R})$ for $\mathcal{R}$ in Example 1

> A semantic definition is a list of axioms and inference rules that define predicates. A semantic definition is identified with a logic, and reasoning with the language is proving theorems within that logic [23, page 23, third paragraph].

We essentially subscribe this point of view, although the "reasoning as theorem proving" part will be revisited.

For instance, the operational description of one-step reduction $\to_{\mathcal{R}}$ in reduction-based systems $\mathcal{R}$ allowing for *conditional rules* $\ell \to r \Leftarrow c$ is naturally made by using inference rules [4, 25, 28, 37, 38]. One-step rewriting is defined as *provability* of goals $s \to t$, where (as in [11]) the usual rewriting symbol $\to$ is viewed as a predicate symbol, in an inference system $\mathcal{I}(\mathcal{R})$. We illustrate this with *Generalized Term Rewriting Systems* (GTRSs [31]) which generalize Conditional Term Rewriting Systems (CTRSs [24]) by enabling the use of *atoms* in the conditions of rules, possibly defined by definite Horn clauses which are part of the GTRS. It is also possible to establish which arguments of each k-ary function symbol f can be rewritten by means of a *replacement map* μ which specifies them as a set $\mu(f) \subseteq \{1,\ldots,k\}$ of *active* arguments [30]. In particular, $\mu_\perp$ *forbids* reductions in all arguments of all function symbols, i.e., $\mu_\perp(f) = \emptyset$.

Example 1 *The GTRS $\mathcal{R} = (\mathcal{F},\Pi,\mu,H,R)$, with $\mathcal{F} = \{\mathsf{0},\mathsf{s}\}$, $\Pi = \{\to,\to^*,\geq,\mathsf{odd},\mathsf{peven},\mathsf{zero}\}$, $\mu = \mu_\perp$, $H = \{(2),(3),(4),(5),(6)\}$, and $R = \{(7)\}$, where:*

$$
\begin{array}{rclr@{\qquad}rclr}
x \geq \mathsf{0} & & & (2) & \mathsf{odd}(x) & \Leftarrow & x \to^* \mathsf{s}(\mathsf{0}) & (5)\\
\mathsf{s}(x) \geq \mathsf{s}(y) & \Leftarrow & x \geq y & (3) & \mathsf{zero}(x) & \Leftarrow & x \to^* \mathsf{0} & (6)\\
\mathsf{peven}(x) & \Leftarrow & x \to^* \mathsf{s}(\mathsf{s}(\mathsf{0})) & (4) & \mathsf{s}(\mathsf{s}(x)) \to x & \Leftarrow & x \geq \mathsf{s}(\mathsf{0}) & (7)
\end{array}
$$

can be used to classify *natural numbers $n \in \mathbb{N}$ written in Peano's notations, i.e., as $\mathsf{s}^n(\mathsf{0})$, into* odd, positive and even*, or* zero *by using predicate symbols* odd, peven*, and* zero*, respectively. Predicate $\geq$ is defined by the Horn clauses (2) and (3); clauses (4), (5), and (6) define the tests; and rule (7) defines one-step rewriting. Computations with $\mathcal{R}$ can be defined by the elementary inference system $\mathcal{I}(\mathcal{R})$ in Figure 1.*

Computational properties of such systems $\mathcal{R}$ are often formulated as *questions* about the relationship between *subject* expressions (e.g., terms $s,t,\ldots$) and the reduction relation $\to_{\mathcal{R}}$ (or some of its extensions and/or combinations: $\to_{\mathcal{R}}^*$, $\to_{\mathcal{R}}^+$, etc.). Expressing such properties as first-order logic formulas is a natural choice. A careful consideration reveals some difficulties, though.

Example 2 *For $\mathcal{R}$ in Example 1 and $\mathcal{I}(\mathcal{R})$ in Figure 1 the following sentence intuitively asserts that every number encoded as a term $\mathsf{s}^n(\mathsf{0})$ for some $n \geq 0$ is odd, or positive and even, or zero:*

$$(\forall x)\,\mathsf{odd}(x) \vee \mathsf{peven}(x) \vee \mathsf{zero}(x) \tag{8}$$

Note that this is true only if x ranges over ground terms t *as above. For instance, if t is a variable x, then there is no proof tree in $\mathcal{I}(\mathcal{R})$ neither for* $\mathsf{odd}(t)$*, nor* $\mathsf{peven}(t)$*, nor* $\mathsf{zero}(t)$*, i.e., (8) does* not *hold.*

Following Clark [6], and different from Kahn (see above), properties of computational systems (e.g., $\mathcal{R}$) expressed as first-order sentences F should be referred to a *canonical* model $\mathcal{M}$ of the theory $\overline{\mathcal{R}}$ describing computations with $\mathcal{R}$. The *choice* of such a model is essential to appropriately understand the property expressed by the formula.

Example 3 *Sentence (8) is satisfied by the usual* least Herbrand model *of $\overline{\mathcal{R}}$ for $\mathcal{R}$ in Example 1 (as $\overline{\mathcal{R}}$ can be seen as a set of Horn clauses, see Figure 3 in Section 3.2), thus fitting the intuitive meaning of the sentence. But also* $\neg(8)$ *is satisfied by Clark's* non-ground least Herbrand model *discussed below, thus disproving the property if x is instantiated to non-ground terms (see Example 41 in Section 5.3).*

This paper investigates the use first-order logic methods, techniques, and tools in the analysis of properties of computational systems defined by means of an EIS so that appropriate solutions to problems like the aforementioned ones can be obtained. Section 2 provides some preliminary definitions; in particular, we remind *Generalized Term Rewriting System* (GTRS [31]) which we often use to illustrate our techniques. In Section 3, borrowing the structure of Smullyan's *Elementary Formal Systems* [46], but using Gentzen's notation for inference rules and deductions [16, 43], we consider inference systems $\mathcal{I}$ consisting of inference rules $\frac{B_1\cdots B_n}{B}$, where $B, B_1, \ldots, B_n$ are *atoms* for some $n \geq 0$, which we call *Elementary Inference Systems* (EISs). As in [46], relations on terms defined by such inference systems are represented by predicate symbols P and obtained by proving atoms $P(t_1, \ldots, t_n)$ in $\mathcal{I}$ by building appropriate *formula-trees* with root $P(t_1, \ldots, t_n)$ (written $\vdash_{\mathcal{I}} P(t_1, \ldots, t_n)$). A (Horn) first-order theory $\mathsf{Th}(\mathcal{I})$ is given to $\mathcal{I}$ so that provable atoms A in $\mathcal{I}$ are characterized as logical consequences of $\mathsf{Th}(\mathcal{I})$. In Section 4 several canonical models are given to $\mathsf{Th}(\mathcal{I})$ so that, in Section 5, properties F expressed as first-order sentences are said to be *semantic properties* of a computational system described by $\mathcal{I}$ *relative to a canonical model $\mathcal{M}$ of* $\mathsf{Th}(\mathcal{I})$ (or just *$\mathcal{M}$-properties* of $\mathcal{I}$) if F is *satisfied* by $\mathcal{M}$, i.e., $\mathcal{M} \models F$ holds. We show how to prove and disprove semantic properties in practice. Section 6 discusses related work. Section 7 concludes.

2 Preliminaries

In the following, we often write *iff* instead of *if and only if*. We assume some familiarity with the basic notions of term rewriting [2, 39, 48] and first-order logic [15, 36].

Given a binary relation $\mathsf{R} \subseteq A \times A$ on a set A, we often write $a\,\mathsf{R}\,b$ instead of $(a,b) \in \mathsf{R}$. The *transitive* closure of R is denoted by R^+, and its *reflexive and transitive* closure by R^*. An element $a \in A$ is *reducible* if there exists b such that $a\,\mathsf{R}\,b$. In this paper, $\mathcal{X}$ denotes a countable set of *variables* and $\mathcal{F}$ denotes a *signature of function symbols*, i.e., a set of *function symbols* $\{f, g, \ldots\}$, each with a fixed *arity* given by a mapping $ar : \mathcal{F} \to \mathbb{N}$. The set of terms built from $\mathcal{F}$ and $\mathcal{X}$ is $\mathcal{T}(\mathcal{F}, \mathcal{X})$; and $\mathcal{T}(\mathcal{F})$ is the set of *ground* terms, i.e., without variable occurrences. The set of variables occurring in t is $\mathcal{V}ar(t)$. We also consider signatures of *predicates* Π. Given a signature $\mathcal{F}$, a *replacement map* is a mapping μ from symbols in $\mathcal{F}$ to sets of positive numbers satisfying $\mu(f) \subseteq \{1, \ldots, ar(f)\}$ for all $f \in \mathcal{F}$ [30].

2.1 First-order logic

Given a signature $\mathcal{F}$ of *function symbols* and a signature Π of *predicate symbols*, atoms $A \in Atoms_{\mathcal{F},\Pi,\mathcal{X}}$ and first-order formulas $F \in Forms_{\mathcal{F},\Pi,\mathcal{X}}$ on such sets of function and predicate symbols with variables

$$(\mathrm{Rf})\quad \frac{}{x \to^* x} \qquad (\mathrm{Pr})_{f,i}\quad \frac{x_i \to y_i}{f(x_1,\ldots,x_i,\ldots,x_k) \to f(x_1,\ldots,y_i,\ldots,x_k)}$$

$$(\mathrm{Co})\quad \frac{x \to y \qquad y \to^* z}{x \to^* z} \qquad (\mathrm{HC})_{B \Leftarrow B_1,\ldots,B_n}\quad \frac{B_1 \quad \cdots \quad B_n}{B}$$

Figure 2: Generic elementary inference rules for a GTRS

in $\mathscr{X}$ are built in the usual way. A (definite) Horn clause (with label α) is written $\alpha : A \Leftarrow A_1,\ldots,A_n$, for atoms $A,A_1,\ldots,A_n$; if $n = 0$, then α is written A rather than $A \Leftarrow$. A first-order theory (FO-theory for short) Th is a set of sentences (formulas whose variables are all *quantified*). An $\mathscr{F},\Pi$-*structure* $\mathscr{A}$ (or just *structure* if no confusion arises) consists of a *non-empty* set $\mathrm{dom}(\mathscr{A})$, called *domain* and often denoted $\mathscr{A}$ if no confusion arises, together with an interpretation of symbols $f \in \mathscr{F}$ and $P \in \Pi$ as mappings $f^{\mathscr{A}}$ and relations $P^{\mathscr{A}}$ on $\mathscr{A}$, respectively. Then, the usual interpretation of first-order formulas with respect to $\mathscr{A}$ is considered [36, page 60]. An $\mathscr{F},\Pi$-model for a theory Th is just a structure $\mathscr{A}$ that makes all the sentences of the theory true, written $\mathscr{A} \models \mathsf{Th}$. A theory Th that has a model is said to be *consistent*. Two theories are *equivalent* if they have the same models. A formula F is a *logical consequence* of a theory Th (written $\mathsf{Th} \models F$) iff every model $\mathscr{A}$ of Th is also a model of F. Also, $\mathsf{Th} \vdash F$ means that F is *deducible* from Th by using a correct and complete deduction procedure.

2.2 Generalized Term Rewriting Systems

A *Generalized Term Rewriting System (GTRS [31, Section 7])* is a tuple $\mathscr{R} = (\mathscr{F},\Pi,\mu,H,R)$ where $\mathscr{F}$ is a signature of *function symbols*, Π is a signature of *predicate symbols*, including at least $\to$ and $\to^*$, $\mu \in M_{\mathscr{F}}$, H is a (possibly empty) set of clauses $A \Leftarrow c$, where $root(A) \notin \{\to,\to^*\}$, and R is a set of rewrite rules $\ell \to r \Leftarrow c$ such that $\ell \notin \mathscr{X}$. In both cases, c is a sequence of atoms. Note that rules in R are Horn clauses.

3 Elementary Inference Systems

In this paper, we consider the following class of inference systems.

Definition 4 (Elementary inference system) *Let $\mathscr{F}$ and Π be signatures of function and predicate symbols, respectively, and $\mathscr{X}$ be a set of variables. An inference rule $\rho : \frac{B_1 \cdots B_n}{B}$ (with label ρ) is called* elementary *if $B,B_1,\ldots,B_n \in Atoms_{\mathscr{F},\Pi,\mathscr{X}}$ are* atoms. *An* elementary *inference system (EIS for short) is a tuple $\mathscr{I} = (\mathscr{F},\Pi,I)$, where I is a set of elementary inference rules.*

Remark 5 *In the literature, inference rules may have a more elaborated structure, typically using* sequents *(usually written $\Gamma \vdash F$, where Γ is an "environment", typically giving values to variables occurring in F, which is an arbitrary formula) instead of just atoms A as components of the rule, see, e.g., [23, Section 2.1]. The structural simplicity of EISs is important to obtain also simple definitions of provability, etc.*

Given an EIS $\mathscr{I} = (\mathscr{F},\Pi,I)$, we often write $\rho \in \mathscr{I}$ instead of $\rho \in I$.

Definition 6 (EIS of a GTRS) *The EIS $\mathscr{I}(\mathscr{R}) = (\mathscr{F},\Pi,I)$ of a GTRS $\mathscr{R} = (\mathscr{F},\Pi,\mu,H,R)$ is (using the generic inference rules in Figure 2):*

$$I = \{(Rf),(Co)\} \cup \bigcup_{f\in\mathscr{F},i\in\mu(f)} \{(Pr)_{f,i}\} \cup \bigcup_{\alpha\in H\cup R} \{(HC)_\alpha\}$$

3.1 Proofs with Elementary Inference Systems

A *finite proof tree* T in $\mathscr{I}$ with root $G \in Atoms_{\mathscr{F},\Pi,\mathscr{X}}$ is either:

- an *open goal*, simply denoted as G; or
- a *derivation tree* denoted as $\frac{T_1 \quad \cdots \quad T_n}{G}(\rho)$, where $T_1,\ldots,T_n$ are finite proof trees in $\mathscr{I}$ (for $n \geq 0$; if $n = 0$ instead of $\frac{}{G}(\rho)$ we just write $\overline{G}$), and $\rho : \frac{B_1 \cdots B_n}{B} \in \mathscr{I}$ is an inference rule such that $G = \sigma(B)$, and $root(T_1) = \sigma(B_1),\ldots,root(T_n) = \sigma(B_n)$ for some substitution σ.

Note that inference rules $\frac{B_1 \cdots B_n}{B}$ in $\mathscr{I}$ are viewed as *schemes of rules* whose head B should *match* the goal G with a matching substitution σ (see [46, Chapter I, #A, §2]). A finite proof tree T is *closed* if it contains no open goals.

Definition 7 (Provable atom) *Let $\mathscr{I}$ be an EIS. An atom A is* provable *in $\mathscr{I}$, written $\vdash_{\mathscr{I}} A$, if there is a closed proof tree T with $root(T) = A$ using $\mathscr{I}$.*

Remark 8 *In the literature, proofs with inference rules may have a more elaborated definition. For instance, the usual rule dealing with the assignment instruction of imperative languages, see, e.g., [42, page 46]:*

$$\frac{\langle e,\varsigma\rangle \longrightarrow^* \langle m,\varsigma\rangle}{\langle v := e,\varsigma\rangle \longrightarrow \varsigma[v \mapsto m]} \tag{9}$$

where e is an expression, ς is a store*, i.e., a mapping from variables to numbers, m is a number, v is a program variable, and $\varsigma[v \mapsto m]$ is a new store obtained from ς so that variable v is bounded to m in $\varsigma[v \mapsto m]$, and any other variable v' different from v remains bounded in $\varsigma[v \mapsto m]$ as it was in ς (see [42, Section 2.1] for the technical details). The* update *$\varsigma[v \mapsto m]$ of a store ς using (9) cannot be handled as the application of a substitution as required by Definition 7. However, if we assume finitely many program variables $v_1,\ldots,v_k$, rule (9) could be seen as k elementary rules as follows:*

$$\frac{\langle e,\mathsf{st}(m_1,\ldots,m_i,\ldots,m_k)\rangle \longrightarrow^* \langle m,\mathsf{st}(m_1,\ldots,m_i,\ldots,m_k)\rangle}{\langle v_i := e,\varsigma\rangle \longrightarrow \mathsf{st}(m_1,\ldots,m,\ldots,m_k)}$$

where $m,m_1,\ldots,m_k$ are variables (disjoint from $v_1,\ldots,v_k$). However, e should be written using indexed variables v_i. Furthermore, evaluation rules for variables should also be decomposed into k rules as follows:

$$\langle v_i,st(m_1,\ldots,m_i,\ldots,m_k)\rangle \longrightarrow \langle m_i,st(m_1,\ldots,m_i,\ldots,m_k)\rangle$$

instead of the (single) Variable rule in [42, page 42], i.e.,

$$\langle v,\varsigma\rangle \longrightarrow \langle \varsigma(v),\varsigma\rangle$$

For each n-ary predicate $P \in \Pi$, the relation on terms $P^{\mathscr{I}}$ defined by $\mathscr{I}$ for P is

$$P^{\mathscr{I}} \quad = \quad \{P(t_1,\ldots,t_n) \mid t_1,\ldots,t_n \in \mathscr{T}(\mathscr{F},\mathscr{X}), \vdash_{\mathscr{I}} P(t_1,\ldots,t_n)\}$$

Provability of (atomic) goals in an EIS is obviously *preserved* under substitution application.

Proposition 9 *Let $\mathscr{I}$ be an EIS, A be an atom, and σ be a substitution. If $\vdash_{\mathscr{I}} A$, then $\vdash_{\mathscr{I}} \sigma(A)$.*

(Rf)	$(\forall x)$	$x \to^* x$	$(\mathrm{HC})_{(4)}$	$(\forall x)$	$x \to^* \mathsf{s}(\mathsf{s}(0)) \Rightarrow \mathsf{peven}(x)$
(Co)	$(\forall x,y,z)$	$x \to y \wedge y \to^* z \Rightarrow x \to^* z$	$(\mathrm{HC})_{(5)}$	$(\forall x)$	$x \to^* \mathsf{s}(0) \Rightarrow \mathsf{odd}(x)$
$(\mathrm{HC})_{(2)}$	$(\forall x)$	$x \geq 0$	$(\mathrm{HC})_{(6)}$	$(\forall x)$	$x \to^* 0 \Rightarrow \mathsf{zero}(x)$
$(\mathrm{HC})_{(3)}$	$(\forall x,y)$	$x \geq y \Rightarrow \mathsf{s}(x) \geq \mathsf{s}(y)$	$(\mathrm{HC})_{(7)}$	$(\forall x)$	$x \geq \mathsf{s}(0) \Rightarrow \mathsf{s}(\mathsf{s}(x)) \to x$

Figure 3: Theory $\overline{\mathcal{R}}$ for $\mathcal{R}$ in Example 1

A finite proof tree T is a *proper prefix* of a finite proof tree T' (written $T \subset T'$) if there are one or more open goals $G_1,\ldots,G_n$ in T such that T' is obtained from T by replacing each G_i by a finite derivation tree T_i with root G_i. An *infinite proof tree* T is an infinite increasing chain of finite proof trees, i.e., a sequence $(T_i)_{i\in\mathbb{N}}$ such that for all i, $T_i \subset T_{i+1}$. Since for all $i \in \mathbb{N}$, $root(T_i) = root(T_{i+1})$, we write $root(T) = root(T_0)$. A finite proof tree T is *well-formed* if it is either an open goal, or a closed proof tree, or a derivation tree $\frac{T_1 \quad \cdots \quad T_n}{G}(\rho)$, where $T_1,\ldots,T_{i-1}$ are closed for some $1 \leq i \leq n$, T_i is a well-formed but not closed finite proof tree, and $T_{i+1},\ldots,T_n$ are open goals. Note the *left-to-right* construction of the proof tree. An infinite proof tree is well-formed if it is an increasing chain of well-formed finite proof trees. As an application of the notion of operational termination [33] we obtain the following.

Definition 10 *(cf. [33, Definition 4]) An EIS $\mathcal{I}$ is called* operationally terminating *if no infinite well-formed proof tree for $\mathcal{I}$ exists.*

In [31, 32], no inference system was given to a GTRS. Only *termination* (of the one-step relation $\to_{\mathcal{R}}$) is discussed in [32]. Using Definition 6, we introduce the following:

Definition 11 *A GTRS $\mathcal{R}$ is operationally terminating if $\mathcal{I}(\mathcal{R})$ is.*

For *binary* predicates $P \in \Pi$, *termination* of the binary relation on terms $P^{\mathcal{I}}$ is defined as expected:

Definition 12 *Let $\mathcal{I} = (\mathcal{F},\Pi,I)$ be an EIS and $P \in \Pi$ be a binary predicate. We say that P is $\mathcal{I}$-terminating if there is no infinite sequence $t_1,t_2,\ldots$ of terms $t_i \in \mathcal{T}(\mathcal{F},\mathcal{X})$ such that, for all $i \geq 1$, $P^{\mathcal{I}}(t_i,t_{i+1})$ holds.*

For GTRSs $\mathcal{R}$, termination of $\mathcal{R}$, i.e., termination of $\to_{\mathcal{R}}$ in the usual sense [31, Section 7.5] coincides with termination of $\to^{\mathcal{I}(\mathcal{R})}$ in Definition 12.

3.2 First-Order Theory of an Elementary Inference System

As done in, e.g., [19, 20], from each elementary inference rule $\rho : \frac{B_1,\ldots,B_n}{B}$ and $\vec{x} = \mathcal{V}ar(B,B_1,\ldots,B_n)$, we obtain a sentence $\overline{\rho}$ (which we call a *definite Horn sentence*) as follows

$$(\forall \vec{x}) \quad B_1 \wedge \cdots \wedge B_n \Rightarrow B$$

If $n = 0$, we just write $(\forall \vec{x})\,B$; if $\vec{x}$ is empty, we just write $B_1 \wedge \cdots \wedge B_n \Rightarrow B$. Given an EIS $\mathcal{I}$, we obtain a theory $\mathsf{Th}(\mathcal{I}) = \{\overline{\rho} \mid \rho \in \mathcal{I}\}$.

Example 13 *For $\mathcal{R}$ in Example 1 and $\mathcal{I}(\mathcal{R})$ in Figure 1, $\overline{\mathcal{R}} = \mathsf{Th}(\mathcal{I}(\mathcal{R}))$ is displayed in Figure 3. By abuse of notation, we use ρ instead of $\overline{\rho}$ to denote sentences $\overline{\rho}$ obtained from inference rules ρ.*

The following result establishes the equivalence between provability of atoms A in an EIS $\mathcal{I}$ and deduction of A (i.e., the universal closure of A) in $\mathsf{Th}(\mathcal{I})$.

Proposition 14 *Let $\mathcal{I}$ be an EIS and A be an atom with variables $\vec{x}$. Then, $\vdash_{\mathcal{I}} A$ iff* $\mathsf{Th}(\mathcal{I}) \vdash (\forall\vec{x})\,A$.

Remark 15 (Provability for GTRSs $\mathcal{R}$) *Since $\mathsf{Th}(\mathcal{I}(\mathcal{R}))$ and the theory $\overline{\mathcal{R}}$ associated to $\mathcal{R}$ in [31, Definition 52] coincide, Proposition 14 shows that defining rewriting steps $s \to_{\mathcal{R}} t$ as deduction of $s \to t$ (i.e., $(\forall\vec{x})\,s \to t$) in $\overline{\mathcal{R}}$ [31, Section 7.5 & Definition 8] is* equivalent *to provability of $s \to t$ in $\mathcal{I}(\mathcal{R})$.*

In the following, for GTRSs we use $\overline{\mathcal{R}}$ rather than $\mathsf{Th}(\mathcal{I}(\mathcal{R}))$.

4 Models of Elementary Inference Systems

Every FO-sentence F can be expressed as a set C_F of clauses (a *standard* form of F [5, Section 4.2]) so that C_F *is inconsistent iff F is* [5, Theorem 4.1]. However, due to *skolemization*, F and C_F are, in general, *not* equivalent.

Example 16 *The set $C_F = \{\mathsf{P}(\mathsf{a})\}$ is a standard form of $F = (\exists x)\mathsf{P}(x)$. The interpretation $\mathcal{A}$ with domain $\mathcal{A} = \{1,2\}$, $\mathsf{a}^{\mathcal{A}} = 1$, and $\mathsf{P}^{\mathcal{A}} = \{(2)\}$ is a model of F but it is* not *a model of C_F [5, page 49].*

Dealing with sets of clauses, we usually consider *Herbrand interpretations*.

4.1 Herbrand interpretations.

The domain of an Herbrand $\mathcal{F},\Pi$-interpretation $\mathcal{H}$ (or just H-interpretation, if no confusion arises) is $\mathrm{dom}(\mathcal{H}) = \mathcal{T}(\mathcal{F})$, which, by the non-emptiness requirement on interpretations (see Section 2), must be *non-empty*; hence $\mathcal{F}$ must contain at least one constant. Each k-ary function symbol $f \in \mathcal{F}$ is given a mapping $f^{\mathcal{A}} : \mathcal{T}(\mathcal{F}) \times \cdots \mathcal{T}(\mathcal{F}) \to \mathcal{T}(\mathcal{F})$ defined by $f^{\mathcal{A}}(t_1,\ldots,t_k) = f(t_1,\ldots,t_k)$ for all $t_1,\ldots,t_k \in \mathcal{T}(\mathcal{F})$. Since the domain and function symbol interpretation are fixed, $\mathcal{H}$ is usually described/identified as a subset $\mathcal{H} \subseteq \mathcal{B}$ of *ground* atoms in the Herbrand Base $\mathcal{B} = Atoms_{\mathcal{F},\Pi,\emptyset}$ [5]. Then, n-ary predicates $P \in \Pi$ are interpreted by $P^{\mathcal{A}} = \{(t_1,\ldots,t_n) \in \mathcal{T}(\mathcal{F})^n \mid P(t_1,\ldots,t_n) \in \mathcal{H}\}$ [5, page 53].

A set of clauses is unsatisfiable (i.e., inconsistent) iff it has no Herbrand model [5, Theorem 4.2]. This may *fail* to hold for arbitrary theories.

Example 17 *Note that* $\mathsf{Th} = \{\mathsf{P}(\mathsf{a}), (\exists x)\neg\mathsf{P}(x)\}$ *is* not *a set of clauses due to the existential quantification of the second formula. It is satisfied by $\mathcal{A}$ with domain $\mathcal{A} = \{0,1\}$, $\mathsf{a}^{\mathcal{A}} = 0$ and $\mathsf{P}^{\mathcal{A}} = \{(0)\}$ but none of the two possible Herbrand interpretations $\mathcal{H}_1 = \emptyset$ and $\mathcal{H}_2 = \{\mathsf{P}(\mathsf{a})\}$ satisfies S [27, pp. 17–18].*

This motivates the following.

Definition 18 (H-consistency) *A theory* Th *is* H-consistent *if it has a Herbrand model. Otherwise, it is* H-inconsistent.

H-consistent theories are consistent, but not vice versa, as Example 17 shows. Furthermore, in sharp contrast to inconsistency, H-inconsistency is *not* preserved by standarization of formulas.

Example 19 *Remind that* $\mathsf{Th} = \{\mathsf{P}(\mathsf{a}), (\exists x)\neg\mathsf{P}(x)\}$ *in Example 17 is H-inconsistent. However, $C_{\mathsf{Th}} = \{\mathsf{P}(\mathsf{a}), \neg\mathsf{P}(c)\}$, where c is a fresh (Skolem) constant, is a standard version of* Th *which is H-consistent as the H-interpretation $\mathcal{H} = \{\mathsf{P}(\mathsf{a})\}$ is a model of C_{Th}.*

The standard semantics for sets of definite Horn clauses over signatures $\mathcal{F}$ and Π of function and predicate symbols (where $\mathcal{F}$ contains at least one constant), using variables in $\mathcal{X}$ [12] considers *Herbrand $\mathcal{F},\Pi$-interpretations* $\mathcal{H}$ viewed as subsets $\mathcal{H} \subseteq \mathcal{B}_{\mathcal{F},\Pi} = Atoms_{\mathcal{F},\Pi,\emptyset}$ of *ground* atoms. We apply these ideas to EIS*s* through $\mathsf{Th}(\mathcal{I})$, which is a set of definite Horn clauses.

4.2 Least Herbrand model of an EIS

Every set of definite Horn clauses has a *least* (with respect to set inclusion) Herbrand $\mathcal{F},\Pi$-model (of ground atomic consequences) [12, Section 5].

Definition 20 (Canonical Herbrand Model of an EIS) *Let $\mathcal{I} = (\mathcal{F},\Pi,I)$ be an EIS. The canonical* H*-model of $\mathcal{I}$ is:*

$$\mathcal{M}(\mathcal{I}) = \{A \in \mathcal{B}_{\mathcal{F},\Pi} \mid \mathsf{Th}(\mathcal{I}) \models A\} = \{A \in \mathcal{B}_{\mathcal{F},\Pi} \mid \mathsf{Th}(\mathcal{I}) \vdash A\} = \{A \in \mathcal{B}_{\mathcal{F},\Pi} \mid \vdash_{\mathcal{I}} A\}$$

Proposition 14 justifies the last equality. As in [12], the *canonicity* of $\mathcal{M}(\mathcal{I})$ comes from the fact that every atom in $\mathcal{M}(\mathcal{I})$ belongs to *every* H-model of $\mathsf{Th}(\mathcal{I})$.

4.3 Least V-Herbrand model of an EIS

Clark extended van Emden and Kowalski's approach to *non-ground (but also called Herbrand) interpretations* $\widehat{\mathcal{H}}$ whose interpretation domain is $\mathcal{T}(\mathcal{F},\mathcal{X})$, rather than $\mathcal{T}(\mathcal{F})$, k-ary function symbols $f \in \mathcal{F}$ are given mappings $f^{\widehat{\mathcal{H}}} : \mathcal{T}(\mathcal{F},\mathcal{X})^k \to \mathcal{T}(\mathcal{F},\mathcal{X})$, and the interpretation of predicate symbols is usually represented as a subset $\widehat{\mathcal{H}} \subseteq \widehat{\mathcal{B}}_{\mathcal{F},\Pi,\mathcal{X}}$ of the non-ground Herbrand base $\widehat{\mathcal{B}}_{\mathcal{F},\Pi,\mathcal{X}} = Atoms_{\mathcal{F},\Pi,\mathcal{X}}$ (or just $\widehat{\mathcal{B}}$ if no confusion arises) consisting of all atoms (possibly with variables). We call them V-Herbrand $\mathcal{F},\Pi$-interpretations, or just $\widehat{\mathsf{H}}$-interpretations. Note that, since $\mathcal{T}(\mathcal{F},\mathcal{X})$ is never empty due to the non-emptiness of $\mathcal{X}$, we do not need to impose that $\mathcal{F}$ contains a constant symbol. As for the standard case, Clark shows the existence of a *least* (with respect to set inclusion) $\widehat{\mathsf{H}}$-model [6, Theorem 3.6]. Accordingly, we introduce the following.

Definition 21 (Canonical V-Herbrand Model of an EIS) *Let $\mathcal{I} = (\mathcal{F},\Pi,I)$ be an EIS. The canonical* $\widehat{\mathsf{H}}$*-model of $\mathcal{I}$ is:*

$$\begin{array}{rcl}\widehat{\mathcal{M}}(\mathcal{I}) & = & \{A \in \widehat{\mathcal{B}}_{\mathcal{F},\Pi,\mathcal{X}} \mid \mathsf{Th}(\mathcal{I}) \models (\forall\vec{x})A\} = \{A \in \widehat{\mathcal{B}}_{\mathcal{F},\Pi,\mathcal{X}} \mid \mathsf{Th}(\mathcal{I}) \vdash (\forall\vec{x})A\} \\ & = & \{A \in \widehat{\mathcal{B}}_{\mathcal{F},\Pi,\mathcal{X}} \mid \vdash_{\mathcal{I}} A\}\end{array}$$

Such a model can be considered as the *canonical* model of the *non-ground* model-theoretic semantics of $\mathcal{I}$. Note that $\mathcal{M}(\mathcal{I}) \subseteq \widehat{\mathcal{M}}(\mathcal{I})$. As we will see in Section 5, having different *canonical* models is essential to define different kind of properties.

For GTRSs $\mathcal{R}$, we write $\mathcal{M}(\mathcal{R})$ and $\widehat{\mathcal{M}}(\mathcal{R})$ rather than $\mathcal{M}(\mathcal{I}(\mathcal{R}))$ and $\widehat{\mathcal{M}}(\mathcal{I}(\mathcal{R}))$. The most natural model for GTRSs is $\widehat{\mathcal{M}}(\mathcal{R})$ as the interpretation domain consists of arbitrary (not only ground) terms, which are the usual 'subject' expressions in term rewriting. However, $\mathcal{M}(\mathcal{R})$ captures important properties as well (see Example 3).

4.4 Grounding the least V-Herbrand model

Let $\mathcal{F},\mathcal{F}'$ and Π,Π' be signatures of function and predicate symbols such that $\mathcal{F} \subseteq \mathcal{F}'$ and $\Pi \subseteq \Pi'$. It is clear that every $\mathcal{F}',\Pi'$-structure $\mathcal{A}$ can be seen as a $\mathcal{F},\Pi$-structure $\mathcal{A}\downarrow_{\mathcal{F},\Pi}$ with the *same* domain of interpretation $\mathrm{dom}(\mathcal{A})$ and taking from $\mathcal{A}$ the interpretations $f^{\mathcal{A}}$ and $P^{\mathcal{A}}$ for all $f \in \mathcal{F}$ and $P \in \Pi$. In the following, we often silently use $\mathcal{F}',\Pi'$-structure as an $\mathcal{F},\Pi$-structure by assuming the previous adaptation.

Let $\mathcal{F}$ be a signature and $\mathcal{X}$ be a denumerable, infinite set of variables such that $\mathcal{F} \cap \mathcal{X} = \emptyset$. Since variables in subject terms t behave like constant symbols in any rewriting sequence, as in, e.g., [1, page 224] and [2, page 78], given a term t, a term $t^{\downarrow}$ is obtained by replacing each occurrence of $x \in \mathscr{X}$ in t by a fresh constant $c_x \notin \mathscr{F} \cup \mathscr{X}$. We let $C_{\mathscr{X}} = \{c_x \mid x \in \mathscr{X}\}$ and $\mathscr{F}_{\mathscr{X}} = \mathscr{F} \cup C_{\mathscr{X}}$. Given a term $t \in \mathscr{T}(\mathscr{F},\mathscr{X})$, its *grounded* version is $t^{\downarrow} \in \mathscr{T}(\mathscr{F}_{\mathscr{X}})$. Vice versa: given $t \in \mathscr{T}(\mathscr{F}_{\mathscr{X}})$, its *ungrounded* version $t^{\uparrow} \in \mathscr{T}(\mathscr{F},\mathscr{X})$ is obtained by replacing, for all $x \in \mathscr{X}$, each constant c_x in t by x. For all terms $t \in \mathscr{T}(\mathscr{F},\mathscr{X})$, $(t^{\downarrow})^{\uparrow} = t$; and for all terms $t \in \mathscr{T}(\mathscr{F}_{\mathscr{X}})$, $(t^{\uparrow})^{\downarrow} = t$. Also, given $A \in Atoms_{\mathscr{F},\Pi,\mathscr{X}}$, $A^{\downarrow} \in Atoms_{\mathscr{F}_{\mathscr{X}},\Pi,\emptyset}$ is its *grounded* version; given $A \in Atoms_{\mathscr{F}_{\mathscr{X}},\Pi,\emptyset}$, $A^{\uparrow} \in Atoms_{\mathscr{F},\Pi,\mathscr{X}}$ is its *ungrounded* version. Given a substitution $\sigma = \{x_1 \mapsto t_1,\ldots,x_n \mapsto t_n\}$, we let $\sigma^{\downarrow} = \{x_1 \mapsto t_1^{\downarrow},\ldots,x_n \mapsto t_n^{\downarrow}\}$. Grounding of variables preserves pattern matching in the following sense.

Proposition 22 *Let $p,t \in \mathscr{T}(\mathscr{F},\mathscr{X})$, $A,B \in Atoms_{\mathscr{F},\Pi,\mathscr{X}}$, and σ be a substitution. Then, (i) $t = \sigma(p)$ iff $t^{\downarrow} = \sigma^{\downarrow}(p)$ and (ii) $A = \sigma(B)$ iff $A^{\downarrow} = \sigma^{\downarrow}(B)$.*

As a consequence of Proposition 22 and the definition of provability in an EIS, we have the following.

Proposition 23 *Let $\mathscr{I} = (\mathscr{F},\Pi,I)$ be an EIS and $A \in Atoms_{\mathscr{F},\Pi,\mathscr{X}}$. Then, $\vdash_{\mathscr{I}} A$ iff $\vdash_{\mathscr{I}} A^{\downarrow}$.*

By Proposition 23, V-Herbrand $\mathscr{F},\Pi$-interpretations $\widehat{\mathscr{H}} \subseteq Atoms_{\mathscr{F},\Pi,\mathscr{X}}$, can be *grounded* into an 'equivalent' Herbrand $\mathscr{F}_{\mathscr{X}},\Pi$-interpretation $\widehat{\mathscr{H}}^{\downarrow} = \{A^{\downarrow} \mid A \in \widehat{\mathscr{H}}\} \subseteq Atoms_{\mathscr{F}_{\mathscr{X}},\Pi,\emptyset}$, which we often call an $\mathsf{H}^{\downarrow}$-interpretation if no confusion arises.

Definition 24 *The* grounded canonical $\mathsf{H}^{\downarrow}$-model $\mathscr{M}^{\downarrow}(\mathscr{I})$ *of $\mathscr{I}$ is $\widehat{\mathscr{M}(\mathscr{I})}^{\downarrow}$.*

Given an EIS $\mathscr{I} = (\mathscr{F},\Pi,I)$, $\mathscr{M}^{\downarrow}(\mathscr{I})$ (viewed as an $\mathscr{F},\Pi$-interpretation) is a model of $\mathsf{Th}(\mathscr{I}) \subseteq Forms_{\mathscr{F},\Pi,\mathscr{X}}$.

Theorem 25 *Let $\mathscr{I} = (\mathscr{F},\Pi,I)$ be an EIS. Then, $\mathscr{M}^{\downarrow}(\mathscr{I}) \models \mathsf{Th}(\mathscr{I})$.*

According to [22, page 39], two $\mathscr{F},\Pi$-structures are *equivalent* if they satisfy the same formulas $F \in Forms_{\mathscr{F},\Pi,\mathscr{X}}$. Then, $\widehat{\mathscr{M}}(\mathscr{I})$ and $\mathscr{M}^{\downarrow}(\mathscr{I})$ are *equivalent*:

Theorem 26 *Let $\mathscr{I} = (\mathscr{F},\Pi,I)$ be an EIS and $F \in Forms_{\mathscr{F},\Pi,\mathscr{X}}$. Then, $\widehat{\mathscr{M}}(\mathscr{I}) \models F$ iff $\mathscr{M}^{\downarrow}(\mathscr{I}) \models F$.*

Theorem 26 justifies that $\mathscr{M}^{\downarrow}(\mathscr{I})$ is called 'canonical' in Definition 24, as it is equivalent (on formulas $F \in Forms_{\mathscr{F},\Pi,\mathscr{X}}$) to the canonical model $\widehat{\mathscr{M}}(\mathscr{I})$. We also have the following "quantifier elimination" results for satisfiability in $\mathscr{M}^{\downarrow}(\mathscr{I})$. In the following, given a term t and set $\mathscr{V}$ of variables, $t^{\downarrow_{\mathscr{V}}}$ is the term obtained by replacing all variables $x \in \mathscr{V}ar(t) \cap \mathscr{V}$ in t by c_x. Similarly for atoms.

Proposition 27 *Let $\mathscr{I} = (\mathscr{F},\Pi,I)$ be an EIS and $A \in Atoms_{\mathscr{F},\Pi,\mathscr{X}}$ be an atom with variables $x_1,\ldots,x_k \in \mathscr{X}$. Then,*

$$\mathscr{M}^{\downarrow}(\mathscr{I}) \models (Q_1x_1)\cdots(Q_kx_k)A \quad \textit{iff} \quad \mathscr{M}^{\downarrow}(\mathscr{I}) \models (\exists x_{\varepsilon_1})\cdots(\exists x_{\varepsilon_p})A^{\downarrow_{\mathscr{V}_U}}$$

where, for all $1 \leq i \leq k$, Q_ix_i represents a quantified variable x_i, where Q_i is a quantifier, either existential ($\exists$) or universal ($\forall$); $E = \{\varepsilon_1,\ldots,\varepsilon_p\}$ is the set of indices of existentially quantified *variables; and $\mathscr{V}_U$ is the set of universally quantified variables.*

Theorem 28 *Let $\mathscr{I}$ be an EIS. Given $n \geq 1$, let $A_1,\ldots,A_n$ be atoms with variables $x_1,\ldots,x_k$, for some $k \geq 0$. Given $m \geq 1$ and $1 \leq n_i \leq m$ for all $1 \leq i \leq m$, let A_{ij} be atoms for all $1 \leq i \leq m$ and $1 \leq j \leq n_i$ with variables $x_1,\ldots,x_k$, for some $k \geq 0$. Let $Q_q \in \{\exists,\forall\}$ for $1 \leq q \leq k$, $E = \{\varepsilon_1,\ldots,\varepsilon_p\} = \{q \mid 1 \leq q \leq k, Q_q = \exists\}$ and $\mathscr{V}_U$ be the set of universally quantified variables. Then,*

$$\mathscr{M}^{\downarrow}(\mathscr{I}) \models (\forall\vec{x})\bigwedge_{i=1}^{n} A_i \quad \textit{iff} \quad \mathscr{M}^{\downarrow}(\mathscr{I}) \models \bigwedge_{j=1}^{n} A_i^{\downarrow} \tag{10}$$

Table 1: Canonical models for Elementary Inference Systems $\mathscr{I} = (\mathscr{F}, \Pi, I)$

Canonical model	Signatures	Type	Atoms in
$\mathscr{M}(\mathscr{I})$	$\mathscr{F}, \Pi$	Herbrand	$Atoms_{\mathscr{F},\Pi,\emptyset}$
$\widehat{\mathscr{M}}(\mathscr{I})$	$\mathscr{F}, \Pi$	V-Herbrand	$Atoms_{\mathscr{F},\Pi,\mathscr{X}}$
$\mathscr{M}^{\downarrow}(\mathscr{I})$	$\mathscr{F}_{\mathscr{X}}, \Pi$	Herbrand	$Atoms_{\mathscr{F}_{\mathscr{X}},\Pi,\emptyset}$

$$\mathscr{M}^{\downarrow}(\mathscr{I}) \models (\forall \vec{x}) \bigvee_{i=1}^{m} \bigwedge_{j=1}^{n_i} A_{ij} \quad \textit{if} \quad \mathscr{M}^{\downarrow}(\mathscr{I}) \models \bigvee_{i=1}^{m} \bigwedge_{j=1}^{n_i} A_{ij}^{\downarrow} \tag{11}$$

$$\textit{If}\; \mathscr{M}^{\downarrow}(\mathscr{I}) \models \overrightarrow{(Q_q x_q)} \bigwedge_{i=1}^{n} A_i, \quad \textit{then} \quad \mathscr{M}^{\downarrow}(\mathscr{I}) \models (\exists x_{\varepsilon_1}) \cdots (\exists x_{\varepsilon_p}) \bigwedge_{i=1}^{n} A_i^{\downarrow \gamma_U} \tag{12}$$

$$\mathscr{M}^{\downarrow}(\mathscr{I}) \models \overrightarrow{(Q_q x_q)} \bigvee_{i=1}^{m} \bigwedge_{j=1}^{n_i} A_{ij} \quad \textit{if} \quad \mathscr{M}^{\downarrow}(\mathscr{I}) \models (\exists x_{\varepsilon_1}) \cdots (\exists x_{\varepsilon_p}) \bigvee_{i=1}^{m} \bigwedge_{j=1}^{n_i} A_{ij}^{\downarrow \gamma_U} \tag{13}$$

Finally, if, for all $j \in E$, x_j occurs in at most one A_i, for some $1 \leq i \leq n$, then

$$\mathscr{M}^{\downarrow}(\mathscr{I}) \models \overrightarrow{(Q_q x_q)} \bigwedge_{i=1}^{n} A_i, \quad \textit{iff} \quad \mathscr{M}^{\downarrow}(\mathscr{I}) \models (\exists x_{\varepsilon_1}) \cdots (\exists x_{\varepsilon_p}) \bigwedge_{i=1}^{n} A_i^{\downarrow \gamma_U} \tag{14}$$

5 Semantic Properties of Elementary Inference Systems

In the following, we adapt the definitions in [29] to the specific setting of EIS.

Definition 29 (Semantic property, cf. [29, Definition 11]) *Let $\mathscr{I} = (\mathscr{F}, \Pi, I)$ be an EIS and $\mathscr{M}$ be an $\mathscr{F}', \Pi'$-model of* $\mathsf{Th}(\mathscr{I})$ *for some $\mathscr{F}' \supseteq \mathscr{F}$ and $\Pi' \supseteq \Pi$ extending $\mathscr{F}$ and Π, respectively. Then, $F \in Forms_{\mathscr{F}',\Pi',\mathscr{X}}$ is a* semantic property *of $\mathscr{I}$ (relative to $\mathscr{M}$, or just an $\mathscr{M}$-property) if $\mathscr{M} \models F$.*

Remark 30 (Use of extended signatures) *In contrast to [29, Definition 11], in Definition 29 we consider extensions $\mathscr{F}'$ and Π' of the original signatures $\mathscr{F}$ and Π of the considered EIS because we consider properties expressed as sentences in $Forms_{\mathscr{F}_{\mathscr{X}},\Pi,\emptyset}$ which must be satisfied in the Herbrand $\mathscr{F}_{\mathscr{X}}, \Pi$-interpretation $\mathscr{M}^{\downarrow}(\mathscr{I})$, as $\mathscr{M}(\mathscr{I})$ and $\widehat{\mathscr{M}}(\mathscr{I})$ provide no interpretation for symbols in $\mathscr{F}_{\mathscr{X}}$.*

Many properties of GTRSs $\mathscr{R}$ can be expressed as *semantic properties* relative to $\widehat{\mathscr{M}}(\mathscr{R})$ (equivalently $\mathscr{M}^{\downarrow}(\mathscr{R})$, see Theorem 26), or $\mathscr{M}(\mathscr{I})$ (for the ground version). In general, such models are *not* comparable regarding their ability to express properties of EISs. Thus, the appropriate choice of a reference model is essential to characterize the targeted property. The *shape* of formulas F also plays a role. We often consider *positive* sentences F of the form:

$$(Q_1 x_1) \cdots (Q_k x_k) \bigwedge_{i=1}^{m} \bigvee_{j=1}^{n_i} A_{ij} \tag{15}$$

where (a) for all $1 \leq i \leq m$ and $1 \leq j \leq n_i$, A_{ij} are *atoms* (which is the reason why we talk of "*positive*" formulas), (b) $x_1, \ldots, x_k$ for some $k \geq 0$ are the variables occurring in those atoms and (c) $Q_1, \ldots, Q_k$ are universal/existential quantifiers. If $Q_i = \exists$ for all $1 \leq q \leq k$, we say that (15) is an *Existentially Closed Boolean Combination of Atoms* (ECBCA for short). We have the following.

Proposition 31 *Let $\mathscr{I} = (\mathscr{F}, \Pi, I)$ be an EIS and $F \in Forms_{\mathscr{F},\Pi,\mathscr{X}}$ be an ECBCA. If $\mathscr{M}(\mathscr{I}) \models F$, then $\widehat{\mathscr{M}}(\mathscr{I}) \models F$ and $\mathscr{M}^{\downarrow}(\mathscr{I}) \models F$.*

Formulas (15) where *only conjunction* is used are called *and*- (or $\wedge$-)formulas.

5.1 Semantic Properties as Logical Consequences

We can prove semantic properties of EIS as logical consequences.

Proposition 32 *(cf. [29, Corollary 14]) Let $\mathscr{I}$ be an EIS and $\mathscr{M}$ be a model of $\mathsf{Th}(\mathscr{I})$. Every logical consequence of $\mathsf{Th}(\mathscr{I})$ is an $\mathscr{M}$-property of $\mathscr{I}$.*

In general, this result *cannot* be reversed [29]. By Proposition 32, we can use theorem provers, e.g., Prover9 [35]) to prove semantic properties, although without distinguishing different (canonical) models.

Example 33 *Term $\mathsf{s}(\mathsf{s}(\mathsf{s}(x)))$ is reducible for arbitrary instances of x to terms in $\mathscr{T}(\mathscr{F}, \mathscr{X})$ if*

$$\mathscr{M}^{\downarrow}(\mathscr{R}) \models (\forall x)(\exists z)\, \mathsf{s}(\mathsf{s}(\mathsf{s}(x))) \to z \tag{16}$$

holds. Since $(\forall x)(\exists z)\, \mathsf{s}(\mathsf{s}(\mathsf{s}(x))) \to z$ is a logical consequence of $\overline{\mathscr{R}}$ (use Prover9), by Proposition 32, (16) holds.

As for Example 1, Proposition 32 cannot be used to prove $\overline{\mathscr{R}} \models (8)$ for $\overline{\mathscr{R}}$ in Figure 3: a *model* of $\overline{\mathscr{R}} \cup \{\neg(8)\}$ can be obtained with, e.g., Mace4 [35], i.e., (8) is not a *logical consequence* of $\overline{\mathscr{R}}$.

5.2 Semantic Properties as Inductive Consequences

For universally quantified positive formulas F we can prove $\mathscr{M}(\mathscr{I}) \models F$ by *induction* on the structure of the set of ground terms $\mathscr{T}(\mathscr{F})$.

Example 34 *For $\mathscr{R}$ in Example 1, we can prove that $\mathscr{M}(\mathscr{R}) \models (8)$ by induction on ground terms t instantiating variable x in (8):*

- *Base case: if $t = \mathsf{0}$, then $\mathsf{zero}(t)$ holds by an application of $(HC)_{(6)}$ using reflexivity rule (Rf).*
- *Induction: let $t = \mathsf{s}^{n+1}(\mathsf{0})$ for some $n \geq 0$ and let $u = \mathsf{s}^n(\mathsf{0})$, i.e., $t = \mathsf{s}(u)$. Assume that (the matrix of) (8) holds on u. We consider three cases:*
    1. *If $\vdash_{\mathscr{I}(\mathscr{R})} \mathsf{zero}(u)$ holds, then, in order to apply $(HC)_{(6)}$, we need either $u = \mathsf{0}$, so that the reflexivity rule (Rf) permits the use of $(HC)_{(6)}$, or else to have $n \div 2 > 0$ applications of $(HC)_{(7)}$ to remove all occurrences of s from u to finally obtain $\mathsf{0}$. Thus, n must be an even number. However, the application of $(HC)_{(7)}$ on u requires that $u = \mathsf{s}(\mathsf{s}(u'))$ and that $u' \geq \mathsf{s}(\mathsf{0})$, which is possible only if n is an odd number. We obtain a contradiction. Thus, it must be $u = \mathsf{0}$ and $t = \mathsf{s}(\mathsf{0})$. We conclude $\vdash_{\mathscr{I}(\mathscr{R})} \mathsf{odd}(t)$ using $(HC)_{(5)}$.*
    2. *If $\vdash_{\mathscr{I}(\mathscr{R})} \mathsf{odd}(u)$ holds, then by reasoning as above, n must be an odd number and hence $n+1$ is a positive even number. We conclude $\vdash_{\mathscr{I}(\mathscr{R})} \mathsf{peven}(t)$ using $(HC)_{(4)}$.*

3. The case when $\vdash_{\mathcal{I}(\mathcal{R})} \mathsf{peven}(u)$ *holds is handled similarly to conclude* $\vdash_{\mathcal{I}(\mathcal{R})} \mathsf{odd}(t)$.

Thus, (the matrix of) (8) holds on t, as desired.

Inductionless induction methods [7, 8] could also be used, as they provide a way to reduce proofs of inductive consequence [7, Definition 2.1] (which implies satisfiability in the least Herbrand model) to proofs of consistency. A set A of first-order formulas is an I-axiomatization of the minimal model $\mathcal{M}(\mathsf{Th})$ of a Horn theory Th if (i) A is a recursive set and contains only purely universal sentences and (ii) $\mathcal{M}(\mathsf{Th})$ is the only Herbrand model of $\mathsf{Th} \cup \mathsf{A}$ up to isomorphism [8, Definition 3]. Then, we have:

Proposition 35 ([8, Proposition 7]) *Let* A *be an I-axiomatization of* $\mathcal{M}(\mathsf{Th})$ *and C be a set of clauses. Then,* $\mathsf{A} \cup \mathsf{Th} \cup C$ *is H-consistent iff* $\mathcal{M}(\mathsf{Th}) \models C$.

In general, Proposition 35 cannot be used with existentially quantified sentences F as the standard clausal form C_F would require skolemization which neither preserve H-consistency (see Example 19) nor satisfiability in a given structure (in this case $\mathcal{M}(\mathsf{Th})$), see Example 16. By [5, Theorem 4.2], consistency and H-consistency are equivalent for clauses. Thus, we have:

Corollary 36 *Let* A *be an I-axiomatization of* $\mathcal{M}(\mathsf{Th})$ *and C be a set of clauses. Then,* $\mathsf{A} \cup \mathsf{Th} \cup C$ *is consistent iff* $\mathcal{M}(\mathsf{Th}) \models C$.

However, obtaining appropriate I-axiomatizations can be difficult.

5.3 Using Satisfiability in Arbitrary Interpretations

Satisfiability in a canonical model can be undecidable (as the membership relation is based on provability or deduction). As in [29], we show how to use satisfaction in arbitrary first-order interpretations $\mathcal{A}$. Given $\mathcal{F},\Pi$-structures $\mathcal{A}$ and $\mathcal{A}'$, a mapping $h : \mathrm{dom}(\mathcal{A}) \to \mathrm{dom}(\mathcal{A}')$ (or just $h : \mathcal{A} \to \mathcal{A}'$ if no confusion arises) is a *homomorphism* if (i) for all k-ary symbols $f \in \mathcal{F}$ and all $a_1,\ldots,a_k \in \mathcal{A}$, $h(f^{\mathcal{A}}(a_1,\ldots,a_k)) = f^{\mathcal{A}'}(h(a_1),\ldots,h(a_k))$ and (ii) for all n-ary predicates $P \in \Pi$ and $a_1,\ldots,a_n \in \mathcal{A}$, if $P^{\mathcal{A}}(a_1,\ldots,a_n)$ holds, then $P^{\mathcal{A}'}(h(a_1),\ldots,h(a_n))$ holds as well [22, Theorem 1.3.1(a) & (b)]. Every model $\mathcal{A}$ of a set $\mathcal{H} \subseteq Atoms_{\mathcal{F},\Pi,\emptyset}$ of *ground atoms* has a *unique* homomorphism $h : \mathcal{T}(\mathcal{F}) \to \mathcal{A}$ [22, Theorem 1.5.1] (the so-called *interpretation homomorphism*). Remind that a mapping $f : D \to E$ is *surjective* if for all $y \in E$ there is $x \in D$ such that $f(x) = y$.

Theorem 37 (Disproving positive $\mathcal{M}(\mathcal{I})$-properties) *(cf. [29, Corollary 28]) Let* $\mathcal{I} = (\mathcal{F},\Pi,I)$ *be an EIS,* $F \in Forms_{\mathcal{F},\Pi,\mathcal{X}}$ *be a positive sentence (15), and* $\mathcal{A}$ *be an* $\mathcal{F},\Pi$*-structure satisfying* $\mathsf{Th}(\mathcal{I}) \cup \{\neg F\}$. *If (i) F is an ECBCA, or (ii)* $h : \mathcal{T}(\mathcal{F}) \to \mathcal{A}$ *is surjective, then* $\mathcal{M}(\mathcal{I}) \models \neg F$ *holds.*

Models $\mathcal{A}$ required in Theorem 37 can often be automatically generated by using model generators like AGES [18] or $\mathsf{Mace4}$ [35].

Example 38 *The following ECBCA represents the existence of a* cycle *in rewriting computations:*

$$(\exists x)(\exists y)\; x \to y \wedge y \to^{*} x \tag{17}$$

We prove that no ground *term starts a cycling reduction with* $\mathcal{R}$ *in Example 1. By Theorem 37.(i), we need to show that there is a model* $\mathcal{A}$ *of* $\overline{\mathcal{R}}$ *which also satisfies* $\neg(17)$. *We use* AGES *to find such a model: the domain is* $\mathcal{A} = \{z \in \mathbb{Z} \mid z \leq 1\}$*; function and predicate symbols are interpreted as follows:*

$$\begin{array}{rclrclrcl}
\mathsf{0}^{\mathcal{A}} & = & 1 & \mathsf{s}^{\mathcal{A}}(x) & = & x-1 & & & \\
\mathsf{odd}^{\mathcal{A}}(x) & \Leftrightarrow & true & \mathsf{peven}^{\mathcal{A}}(x) & \Leftrightarrow & true & \mathsf{zero}^{\mathcal{A}}(x) & \Leftrightarrow & true \\
x \geq^{\mathcal{A}} y & \Leftrightarrow & true & x \to^{\mathcal{A}} y & \Leftrightarrow & y > x & x(\to^{*})^{\mathcal{A}} y & \Leftrightarrow & y \geq x
\end{array}$$

Surjectivity of $h : \mathscr{T}(\mathscr{F}) \to \mathscr{A}$ (required in Theorem 37(ii)) can be guaranteed by using an appropriate theory SuH [29, Section 6]. For instance, given a non-empty, finite set $T \subseteq \mathscr{T}(\mathscr{F})$ of ground terms and

$$\mathsf{SuH}^T = \{(\forall x) \bigvee_{t \in T} x = t\}$$

by [29, Proposition 40], $\mathscr{A} \models \mathsf{SuH}^T$ implies surjectivity of h. A more general approach is described in [29, Section 6.2].

Formulas F involving symbols in $C_{\mathscr{X}}$ cannot be proved as semantic properties w.r.t. $\mathscr{M}(\mathscr{I})$ because symbols in $C_{\mathscr{X}}$ are not interpreted by $\mathscr{M}(\mathscr{I})$. Instead, $\mathscr{M}^{\downarrow}(\mathscr{I})$ should be used. However, $\mathscr{M}^{\downarrow}(\mathscr{I})$ is an $\mathscr{F}_{\mathscr{X}}, \Pi$-structure. Hence, $\mathscr{A}$ should be an $\mathscr{F}_{\mathscr{X}}, \Pi$-structure to be able to use Theorem 37 applied to $\mathscr{F}_{\mathscr{X}}$. However, $\mathscr{F}_{\mathscr{X}}$ is infinite (due to infiniteness of $\mathscr{X}$), and synthesizing structures $\mathscr{A}$ interpreting infinitely many symbols can be difficult. Since F contains a *finite* (possibly empty) set of symbols $K \subseteq C_{\mathscr{X}}$, and $\mathsf{Th}(\mathscr{I}) \subseteq Forms_{\mathscr{F},\Pi,\mathscr{X}}$, we can try to use $\mathscr{F} \cup K, \Pi$-structures $\mathscr{A}$ instead.

Theorem 39 (Disproving positive $\mathscr{M}^{\downarrow}(\mathscr{I})$-properties) *Let $\mathscr{I} = (\mathscr{F}, \Pi, I)$ be an EIS, $\mathscr{X}$ be a set of variables, $K \subseteq \mathscr{F}_{\mathscr{X}}$, and $F \in Forms_{\mathscr{F} \cup K,\Pi,\mathscr{X}}$ be a positive sentence (15), and $\mathscr{A}$ be an $\mathscr{F} \cup K, \Pi$-model of $\mathsf{Th}(\mathscr{I})$. If (i) F is an ECBCA and $\mathscr{A} \models \neg F$ holds, or (ii) F is an $\wedge$-positive formula and U is the set of universally quantified variables in F and $\mathscr{A} \models \neg F^{\downarrow_U}$ holds or (iii) $h : \mathscr{T}(\mathscr{F} \cup K) \to \mathscr{A}$ is surjective and $\mathscr{A} \models \neg F$ holds, then $\mathscr{M}^{\downarrow}(\mathscr{I}) \models \neg F$ holds.*

Remark 40 (Formulas $F \in Forms_{\mathscr{F},\Pi,\mathscr{X}}$ without grounded variables) *If F contains no grounded variables c_x, then K in Theorem 39 can be taken as* empty. *In this case, proving that $\mathscr{M}^{\downarrow}(\mathscr{I}) \models \neg F$ holds using items (i) and (iii) in Theorem 39 would also prove $\mathscr{M}(\mathscr{I}) \models \neg F$ as those items would coincide with the conditions of use of Theorem 37. However, it may happen that $\mathscr{M}(\mathscr{I}) \models F$ holds but $\mathscr{M}^{\downarrow}(\mathscr{I}) \models F$ does* not *hold (see Example 2 and Example 41 below). In this case, with $K = \emptyset$, Theorem 39 could* not *be used to conclude $\mathscr{M}^{\downarrow}(\mathscr{I}) \models \neg F$. Then, we let $K \neq \emptyset$ so that Theorem 39 can be advantageously used.*

Example 41 *We prove that $\mathscr{M}^{\downarrow}(\mathscr{R}) \models \neg(8)$ holds by using Theorem 39.(iii). Let $K = \{c_x\}$ and $T = \{0, c_x\}$. Hence, $\mathsf{SuH}^T = \{(\forall x)\, x = 0 \vee x = c_x\}$. We obtain a model $\mathscr{A}$ of*

$$\overline{\mathscr{R}} \cup \mathsf{SuH}^T \cup \{\neg(\forall x)(\mathsf{peven}(x) \vee \mathsf{odd}(x)) \vee \mathsf{zero}(x))\}$$

with Mace4. *The domain is $\{0,1\}$; the interpretations of function symbols is*

$$\mathsf{c_x}^{\mathscr{A}} = 1 \qquad 0^{\mathscr{A}} = 1 \qquad \mathsf{s}^{\mathscr{A}}(x) = x+1$$

and all predicate symbols (except the equality symbol) are interpreted as true.

6 Related work

Our elementary inference systems combine aspects of Smullyan's *Elementary Formal Systems* and Mathematical Systems [46, Chapter 1, #A, §1 and §4] (emphasizing the idea of *defining* sets or relations by deduction using implicative (schemes of) axioms $B_1 \Rightarrow B_2 \Rightarrow \cdots \Rightarrow B_n \Rightarrow B$, where $B_1, \ldots, B_n$ and B are *atoms*) and Gentzen's notion of *inference rules* (where such implicative axioms are displayed as inference rules $\frac{B_1 \cdots B_n}{B}$) and the arrangement of deductions as *formula-trees* [43, Chapter 1, §2, B], which is essential to make sense of the notion of *operational (non-)termination*, which cannot be captured by using Smullyan's notion of deduction of atoms in an elementary formal system. On the other hand, Gentzen's general notion of inference rule $\frac{F_1 \cdots F_n}{F}$ (or *inference figure* in his terminology) permits the use of arbitrary formulas $F_1, \ldots, F_n$ and F in the upper and lower parts of the inference rule [16, Section I, item 3.1], thus obtaining *more general* inference rules than Smullyan's and ours. Both Smullyan and Prawitz emphasize the use of *instances* of inference rules in deduction (rather than the explicit inclusion of substitutions in rules, as in [4, 25]) a keypoint which we follow in our definitions and methods.

After describing a computational system as a *first-order theory* Th, the use of first-order sentences F to express properties of a computational system (programming language, database, etc.) is a natural choice [17, 34], and a *"properties-as-logical-consequences"* approach has been frequently adopted to claim/deny the property of the considered system [17]. Clark's approach, however, is that sentences expressing properties should be checked *with respect to a given canonical model only* [6, Chapter 4]. After the seminal work on the model-theoretic description of the semantics of logic programming [12], other approaches have been proposed, including the use of non-ground Herbrand interpretations [6] and other refinements [13, 14, 26]. In the realm of Term Rewriting Systems, a different path has been followed using the *first-order theory of rewriting* (*FOThR*) for TRSs $\mathcal{R}$ [10], where predicate symbols $\to$ and $\to^*$ are *interpreted* on the least Herbrand model $\mathcal{M}(\mathcal{R})$ of $\overline{\mathcal{R}}$. However, only formulas F containing *no constant or function symbol* can be used to express properties which are checked by satisfiability in $\mathcal{M}(\mathcal{R})$ [9, Section 6]. For instance, *ground confluence* of rewriting computations is expressed as follows:

$$(\forall x, y, z)\ (x \to^* y \wedge x \to^* z \Rightarrow (\exists u)(y \to^* u \wedge z \to^* u)) \tag{18}$$

and $\mathcal{M}(\mathcal{R}) \models (18)$ means that $\mathcal{R}$ is ground confluent, as variables in (18) range on *ground* terms (the Herbrand Universe) only. Tree automata techniques can be used to prove properties of *ground* TRSs $\mathcal{R}$. Recently, the approach was extended to left-linear, right-ground TRSs [44]. The tool Fort [45] provides an implementation. In contrast, we are able to deal with GTRSs and properties can be expressed in a more flexible way. For instance, among the properties considered above, only non-cyclingness of $\mathcal{R}$ (17) can be expressed in FOThR; however, the results in [10, 44] does not apply to prove it of $\mathcal{R}$ in Example 1.

7 Conclusions and Future Work

Borrowing Smullyan's *elementary formal systems* using Gentzen's notation for inference rules we have introduced *Elementary Inference Systems* (EISs) $\mathcal{I}$, consisting of (elementary) inference rules $\frac{B_1, \ldots, B_n}{B}$ where B, $B_1, \ldots, B_n$ are atoms. Sets, relations, and computations can be defined by associating a proof-tree to a given atom $A = P(t_1, \ldots, t_n)$ which is *matched* by the lower part B of an inference rule $\frac{B_1, \ldots, B_n}{B}$, i.e., $A = \sigma(B)$ for some substitution σ, provided that the corresponding instances $\sigma(B_i)$ of each B_i, $1 \leq i \leq n$ can also be proved analogously. A first-order (Horn) theory $\mathsf{Th}(\mathcal{I})$ is given to $\mathcal{I}$ so that atoms A that can be proved in $\mathcal{I}$ can be *deduced* from $\mathsf{Th}(\mathcal{I})$ and vice versa. Also, canonical (Herbrand or V-Herbrand) models $\mathcal{M}(\mathcal{I})$, $\widehat{\mathcal{M}}(\mathcal{I})$, and $\mathcal{M}^{\downarrow}(\mathcal{I})$ of $\mathsf{Th}(\mathcal{I})$ are given to $\mathcal{I}$ so that properties of $\mathcal{I}$ expressed as first-order sentences F can often be proved of $\mathcal{I}$ by *satisfaction* in the corresponding canonical models. We call them *semantic properties* of $\mathcal{I}$. Practical and mechanizable approaches to prove semantic properties, including the use of theorem provers and model generation tools like AGES, $\mathsf{Mace4}$, and $\mathsf{Prover9}$, have been illustrated by means of examples showing their use in the analysis of semantic properties of GTRSs. In the future, we intend to give direct support in AGES to the techniques described in this paper.

Acknoledgements. I thank the anonymous reviewers for their useful comments and suggestions.

References

[1] Jürgen Avenhaus & Carlos Loría-Sáenz (1994): *On Conditional Rewrite Systems with Extra Variables and Deterministic Logic Programs*. In Frank Pfenning, editor: *Logic Programming and Automated Reasoning, 5th International Conference, LPAR'94, Proceedings*, *Lecture Notes in Computer Science* 822, Springer, pp. 215–229, doi:10.1007/3-540-58216-9_40.

[2] Franz Baader & Tobias Nipkow (1998): *Term Rewriting and All That*. Cambridge University Press, doi:10.1017/CBO9781139172752.

[3] Hendrik Pieter Barendregt (1971): *Some extensional term models for combinatory logics and λ-calculi*. Ph.D. thesis, University of Utrecht.

[4] Roberto Bruni & José Meseguer (2006): *Semantic foundations for generalized rewrite theories*. *Theor. Comput. Sci.* 360(1-3), pp. 386–414, doi:10.1016/j.tcs.2006.04.012.

[5] Chin-Liang Chang & Richard C. T. Lee (1973): *Symbolic logic and mechanical theorem proving*. Computer science classics, Academic Press.

[6] Keith L. Clark (1980): *Predicate logic as a computational formalism*. Ph.D. thesis, Queen Mary University of London, UK. Available at `http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.253345`.

[7] Hubert Comon (2001): *Inductionless Induction*. In John Alan Robinson & Andrei Voronkov, editors: *Handbook of Automated Reasoning (in 2 volumes)*, Elsevier and MIT Press, pp. 913–962, doi:10.1016/B978-044450813-3/50016-3.

[8] Hubert Comon & Robert Nieuwenhuis (2000): *Induction=I-Axiomatization+First-Order Consistency*. *Inf. Comput.* 159(1-2), pp. 151–186, doi:10.1006/INCO.2000.2875.

[9] Max Dauchet (1993): *Rewriting and Tree Automata*. In Hubert Comon & Jean-Pierre Jouannaud, editors: *Term Rewriting, French Spring School of Theoretical Computer Science, Font Romeux, France, May 17-21, 1993, Advanced Course*, *Lecture Notes in Computer Science* 909, Springer, pp. 95–113, doi:10.1007/3-540-59340-3_8.

[10] Max Dauchet & Sophie Tison (1990): *The Theory of Ground Rewrite Systems is Decidable*. In: *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990*, IEEE Computer Society, pp. 242–248, doi:10.1109/LICS.1990.113750.

[11] Joëlle Despeyroux (1986): *Proof of Translation in Natural Semantics*. In: *Proceedings of the Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18, 1986*, IEEE Computer Society, pp. 193–205.

[12] Maarten H. van Emden & Robert A. Kowalski (1976): *The Semantics of Predicate Logic as a Programming Language*. *J. ACM* 23(4), pp. 733–742, doi:10.1145/321978.321991.

[13] Moreno Falaschi, Giorgio Levi, Maurizio Martelli & Catuscia Palamidessi (1993): *A Model-Theoretic Reconstruction of the Operational Semantics of Logic Programs*. *Inf. Comput.* 103(1), pp. 86–113, doi:10.1006/INCO.1993.1015.

[14] Moreno Falaschi, Giorgio Levi, Catuscia Palamidessi & Maurizio Martelli (1989): *Declarative Modeling of the Operational Behavior of Logic Languages*. *Theor. Comput. Sci.* 69(3), pp. 289–318, doi:10.1016/0304-3975(89)90070-4.

[15] Melvin Fitting (1996): *First-Order Logic and Automated Theorem Proving, Second Edition*. Graduate Texts in Computer Science, Springer, doi:10.1007/978-1-4612-2360-3.

[16] Gerhard Gentzen (1935): *Untersuchungen über das logische schliessen, I*. *Mathematische Zeitschrift* 39, pp. 176–210, doi:10.1007/BF01201353. English version in [47, pages 68-131].

[17] C. Cordell Green & Bertram Raphael (1968): *The Use of Theorem-Proving Techniques in Question-Answering Systems*. In: *Proceedings of the 1968 23rd ACM National Conference*, ACM '68, Association for Computing Machinery, New York, NY, USA, p. 169–181, doi:10.1145/800186.810578.

[18] Raúl Gutiérrez & Salvador Lucas (2019): *Automatic Generation of Logical Models with AGES*. In Pascal Fontaine, editor: *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Proceedings*, *Lecture Notes in Computer Science* 11716, Springer, pp. 287–299, doi:10.1007/978-3-030-29436-6_17.

[19] John Hannan & Dale Miller (1989): *Deriving Mixed Evaluation from Standard Evaluation for a Simple Functional Language*. In Jan L. A. van de Snepscheut, editor: *Mathematics of Program Construction, 375th Anniversary of the Groningen University, International Conference, Groningen, The Netherlands, June 26-30, 1989, Proceedings*, *Lecture Notes in Computer Science* 375, Springer, pp. 239–255, doi:10.1007/3-540-51305-1_13.

[20] John Hannan & Dale Miller (1990): *From Operational Semantics to Abstract Machines: Preliminary Results*. In Gilles Kahn, editor: *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*, ACM, pp. 323–332, doi:10.1145/91556.91680.

[21] J. Roger Hindley & Jonathan P. Seldin (1986): *Introduction to Combinators and Lambda-Calculus*. Cambridge University Press.

[22] Wilfrid Hodges (1997): *A Shorter Model theory*. Cambridge University Press.

[23] Gilles Kahn (1987): *Natural Semantics*. In Franz-Josef Brandenburg, Guy Vidal-Naquet & Martin Wirsing, editors: *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*, *Lecture Notes in Computer Science* 247, Springer, pp. 22–39, doi:10.1007/BFB0039592.

[24] Stéphane Kaplan (1984): *Conditional Rewrite Rules*. *Theor. Comput. Sci.* 33, pp. 175–193, doi:10.1016/0304-3975(84)90087-2.

[25] René Lalement (1993): *Computation as logic*. Prentice Hall International series in computer science, Prentice Hall.

[26] Giorgio Levi & Catuscia Palamidessi (1985): *The Declarative Semantics of Logical Read-Only Variables*. In: *Proceedings of the 1985 Symposium on Logic Programming, Boston, Massachusetts, USA, July 15-18, 1985*, IEEE-CS, pp. 128–137.

[27] John W. Lloyd (1987): *Foundations of Logic Programming, 2nd Edition*. Springer, doi:10.1007/978-3-642-83189-8.

[28] Salvador Lucas (2017): *Analysis of Rewriting-Based Systems as First-Order Theories*. In Fabio Fioravanti & John P. Gallagher, editors: *Logic-Based Program Synthesis and Transformation - 27th International Symposium, LOPSTR 2017, Namur, Belgium, October 10-12, 2017, Revised Selected Papers*, *Lecture Notes in Computer Science* 10855, Springer, pp. 180–197, doi:10.1007/978-3-319-94460-9_11.

[29] Salvador Lucas (2019): *Proving semantic properties as first-order satisfiability*. *Artif. Intell.* 277, doi:10.1016/j.artint.2019.103174.

[30] Salvador Lucas (2020): *Context-sensitive Rewriting*. *ACM Comput. Surv.* 53(4), pp. 78:1–78:36, doi:10.1145/3397677.

[31] Salvador Lucas (2024): *Local confluence of conditional and generalized term rewriting systems*. *Journal of Logical and Algebraic Methods in Programming* 136, pp. paper 100926, pages 1–23, doi:10.1016/j.jlamp.2023.100926.

[32] Salvador Lucas (2024): *Termination of Generalized Term Rewriting Systems*. In Jakob Rehof, editor: *9th International Conference on Formal Structures for Computation and Deduction (FSCD 2024)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 299, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 29:1–29:18, doi:10.4230/LIPIcs.FSCD.2024.29.

[33] Salvador Lucas, Claude Marché & José Meseguer (2005): *Operational termination of conditional term rewriting systems*. *Inf. Process. Lett.* 95(4), pp. 446–453. Available at `http://dx.doi.org/10.1016/j.ipl.2005.05.002`.

[34] Zohar Manna (1969): *Properties of Programs and the First-Order Predicate Calculus*. *J. ACM* 16(2), pp. 244–255, doi:10.1145/321510.321516.

[35] William McCune (2005–2010): *Prover9 & Mace4.* Technical Report, University of New Mexico. Available at `http://www.cs.unm.edu/~mccune/prover9/`.

[36] Elliott Mendelson (1997): *Introduction to mathematical logic (4. ed.).* Chapman and Hall.

[37] José Meseguer (1992): *Conditional Rewriting Logic as a Unified Model of Concurrency*. *Theor. Comput. Sci.* 96(1), pp. 73–155, doi:10.1016/0304-3975(92)90182-F.

[38] José Meseguer (2012): *Twenty years of rewriting logic*. *J. Log. Algebr. Program.* 81(7-8), pp. 721–781, doi:10.1016/j.jlap.2012.06.003.

[39] Enno Ohlebusch (2002): *Advanced Topics in Term Rewriting*. Springer, doi:10.1007/978-1-4757-3661-8.

[40] Gordon D. Plotkin (1981): *A structural approach to operational semantics*. Technical Report DAIMI FN-19, Computer Science Department. Aarhus University.

[41] Gordon D. Plotkin (2004): *The origins of structural operational semantics*. *J. Log. Algebraic Methods Program.* 60-61, pp. 3–15, doi:10.1016/J.JLAP.2004.03.009.

[42] Gordon D. Plotkin (2004): *A structural approach to operational semantics*. *J. Log. Algebraic Methods Program.* 60-61, pp. 17–139.

[43] Dag Prawitz (1965): *Natural deduction. A proof theoretical study*. Stockholm Studies in Philosophy, Almqvist & Wiksell.

[44] Franziska Rapp & Aart Middeldorp (2016): *Automating the First-Order Theory of Rewriting for Left-Linear Right-Ground Rewrite Systems*. In Delia Kesner & Brigitte Pientka, editors: *1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, June 22-26, 2016, Porto, Portugal*, *Leibniz International Proceedings in Informatics (LIPIcs)* 52, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 36:1–36:12, doi:10.4230/LIPIcs.FSCD.2016.36.

[45] Franziska Rapp & Aart Middeldorp (2018): *FORT 2.0*. In Didier Galmiche, Stephan Schulz & Roberto Sebastiani, editors: *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Proceedings*, *Lecture Notes in Computer Science* 10900, Springer, pp. 81–88, doi:10.1007/978-3-319-94205-6_6.

[46] Raymond Smullyan (1961): *Theory of Formal Systems*. Princeton University Press.

[47] Manfred E. Szabo (1969): *The Collected Papers of Gerhard Gentzen*. *Studies in Logic and the Foundations of Mathematics* 55, Elsevier, doi:10.1016/S0049-237X(08)70822-X.

[48] Terese (2003): *Term rewriting systems*. *Cambridge tracts in theoretical computer science* 55, Cambridge University Press.

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.

Funding. This research was partially funded by the 2024-2.1.1-EKOP-2024-00003 University Research Scholarship Programme under project numbers EKOP-24-3-BME-{78,213}, and the Doctoral Excellence Fellowship Programme under project number 400434/2023; with the support provided by the Ministry of Culture and Innovation of Hungary from the NRDI Fund.

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`

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

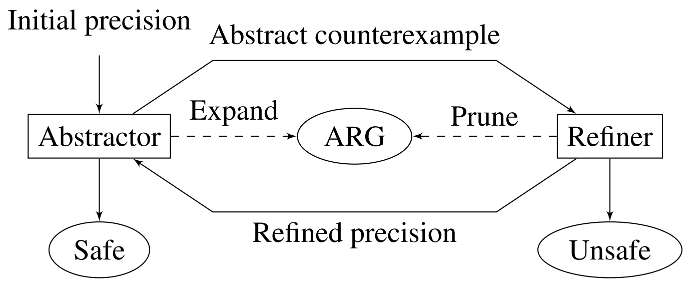


Figure 1: The CEGAR loop

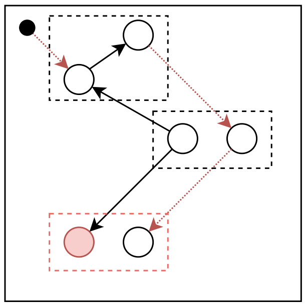


Figure 2: An ARG

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 $a = 2 \wedge b = 3$ 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 alleviate 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.

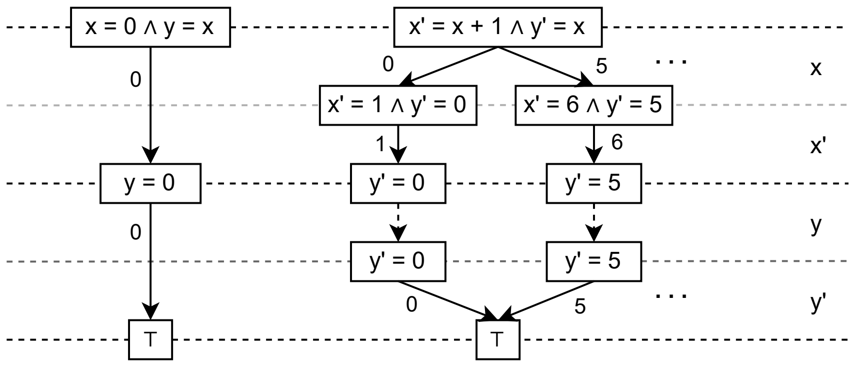


Figure 3: Substitution diagram characterizing the initial states $I : x = 0 \wedge y = x$ (left) and 2k-level substitution diagram characterizing the transition relation $T : x' = x + 1 \wedge y' = x$ (right). Each level has an associated variable, nodes correspond to SMT formulas, while the edges represent 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 $\perp$ 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 $(loc = inv) \implies x = y + 2 \land y = z + 1$ will be transformed into the model

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

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.

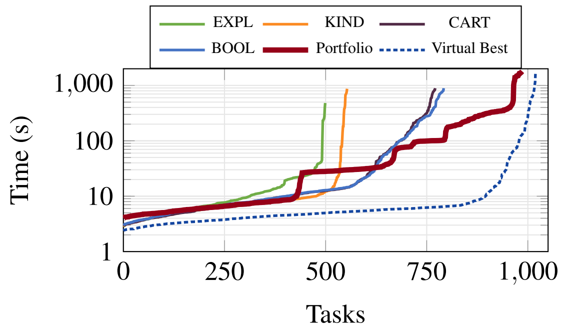
In order to answer RQ2, we counted the number of tasks a configuration solved in a category (keeping in mind, that a prior successful configuration disables a later configuration, so we naturally expect diminished results the later a configuration is). These results are shown in Table 2. Note that these 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.

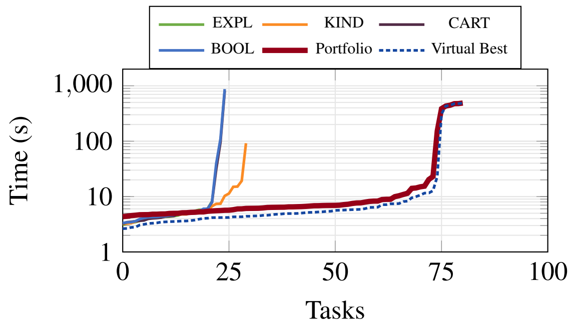
4.1 Analysis of the Results

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.

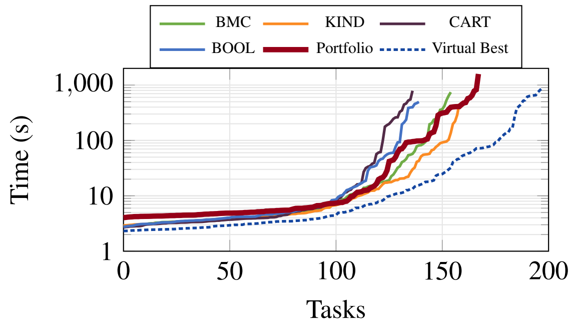
[3]https://github.com/chc-comp/chc-comp25-benchmarks/commit/ceec2f7740478cc9f114aa87fc54fc167738acdf



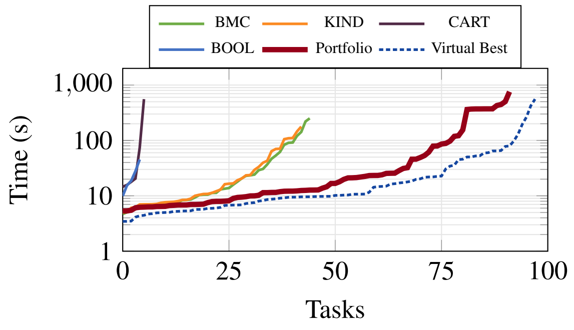
(a) LIA-Lin



(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: FMCAD (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

CHCVerif: A Portfolio-Based Solver for Constrained Horn Clauses

Mihály Dobos-Kovács   Levente Bajczi   András Vörös

Department of Artificial Intelligence and Systems Engineering
Budapest University of Technology and Economics, Hungary

{mdobosko,bajczi,vori}@mit.bme.hu

Constrained Horn Clauses (CHCs) are widely adopted as intermediate representations for a variety of verification tasks, including safety checking, invariant synthesis, and interprocedural analysis. This paper introduces CHCVerif, a portfolio-based CHC solver that adopts a software verification approach for solving CHCs. This approach enables us to reuse mature software verification tools to tackle CHC benchmarks, particularly those involving bitvectors and low-level semantics. Our evaluation shows that while the method enjoys only moderate success with linear integer arithmetic, it achieves modest success on bitvector benchmarks. Moreover, our results demonstrate the viability and potential of using software verification tools as backends for CHC solving, particularly when supported by a carefully constructed portfolio.

Funding. This research was partially funded by the 2024-2.1.1-EKOP-2024-00003 University Research Scholarship Programme under project numbers EKOP-24-3-BME-213, and the Doctoral Excellence Fellowship Programme under project numbers DKÖP-23-1-BME-5 and DKÖP-23-1-BME-15; with the support provided by the Ministry of Culture and Innovation of Hungary from the NRDI Fund.

1 Introduction

Constrained Horn Clauses (CHCs) form a widely adopted intermediate representation for a variety of verification tasks, including safety checking [29, 16, 18], invariant synthesis [21, 23, 17], and systems verification [12]. Consequently, CHC solving has become a fundamental component in many formal verification pipelines. Despite the maturity of existing solvers such as Eldarica [20] and Spacer [24], there are still unsolved challenges, such as supporting CHCs with bitvector theories.

We present CHCVerif, a portfolio-based CHC solver that adopts a software verification perspective. Rather than solving CHCs directly at the logical level, CHCVerif translates CHC problems into semantically equivalent C programs and applies a portfolio of C verifiers to determine their correctness. The satisfiability of the original CHC system corresponds to the correctness of the generated C program. This approach enables CHCVerif to take advantage of mature software verification backends while diversifying its solving strategy through portfolio composition.

CHCVerif builds on CoVeriTeam [4], a recently proposed framework for constructing compositional verification portfolios. CoVeriTeam allows CHCVerif to orchestrate multiple C verifiers within a configurable and extensible architecture. By expressing CHC solving as a software verification task and managing diverse tools through CoVeriTeam, CHCVerif offers a novel and modular pathway to Horn clause verification.

The remainder of the paper is structured as follows. Section 2 provides background on portfolio-based verification. Section 3 describes our overall approach to verification. Section 4 presents our process of constructing the portfolio. In Section 5, we reflect on our results.

2 Background

2.1 CHC to C transformation

Let us define a CHC problem as a set of *deduction rules* in the context of this work. A deduction rule consists of a premise and a consequence. The premise can have zero or more *uninterpreted functions*, while the consequence can have zero or exactly one.

If there are no uninterpreted functions in the premise, we call that rule an *atom*. If there is more than one uninterpreted function in the premise of a rule, we call that rule *non-linear*. If a deduction rule deduces the literal *false*, we call that rule a *query*.

If any rule is non-linear in a CHC problem, we call that problem *non-linear*. Otherwise, we say that the CHC is *linear*.

In practical applications, CHC problems are frequently represented using the SMT-LIBv2 format [11, 13]. This format depicts deduction rules as *imply* expressions over a specific SMT theory. Moreover, every variable involved in the deduction rules is subject to universal quantification across the entire domain of variables, as required by the respective SMT theories.

In this paper, we focus on the SMT theories *core*, *linear integer arithmetic*, and *fixed-size bitvectors*. Note that problems requiring support for more theories exist (such as those using *arrays* or *algebraic data types*), but we discount those in the context of this work.

In the following, we present a simple example to showcase the basic idea behind the CHC to C transformation from [26, 2].

$$\begin{aligned} A(x) &\leftarrow x = 1 \\ A(x) &\leftarrow A(x-1) \\ \mathit{false} &\leftarrow A(11) \end{aligned}$$

The first rule declares that $A(1)$ is *true*. The second rule propagates this fact forward, stating that if $A(x)$ holds, then $A(x+1)$ must also hold. Finally, the third rule is a *query* that leads to a contradiction if $A(11)$ holds. Intuitively, this system is unsatisfiable: by repeated application of the second rule, starting from $A(1)$, we deduce $A(2)$, $A(3)$, and so on up to $A(11)$. Thus, $A(11)$ is deduced, which triggers the contradiction, making the system unsatisfiable.

The CHC system can be encoded in software in a top-down or bottom-up manner [26]. In the backward (top-down) encoding in Listing 1, the verifier starts with the query $A(11)$ and recursively unfolds the rules until it reaches the exit condition. Note that this program might fall into an infinite recursion based on the value of the parameter of the function A. However, tools that can reason about recursion may find that the exit condition of $A(11)$ is reachable.

In contrast, the forward (bottom-up) version shown in Listing 2 constructs the program starting from the facts. It iteratively applies the rules using a loop and a nondeterministic choice of the current state. By explicitly modeling the inference steps, this version avoids recursion and can often be more amenable to automated reasoning by tools that support nondeterminism and state exploration, but in return, it only works with linear CHCs.

2.2 Algorithm selection and portfolios

Formal verification, in general, is a resource-intensive task that involves a variety of analysis techniques, such as symbolic execution, bounded model checking, abstract interpretation, and others, each with

Listing 1: Backward transformation

```c
int A(int x) {
    if(x == 1) return 1;
    else if(A(x - 1)) return 1;
    else return 0;
}

int main() {
    if(A(11)) return -1;
    else return 0;
}
```

Listing 2: Forward transformation

```c
int main() {
    int A = 1, x;

    while(true) {
        x = nondet();
        if(A == 11) return -1;
        else if(A == x - 1) A = x;
    }
}
```

its own strengths and limitations. Given the inherent variability in how different tools perform across verification problems, using portfolio-based and algorithm selection strategies has received increasing attention in recent years [7].

In the context of formal verification, the task of algorithm selection involves determining the most suitable algorithm, configuration, or tool from a set of options to solve a verification problem. The choice is typically based on characteristics derived from the problem, such as its structural details, control flow, data types, or recognizable patterns.

In contrast, a portfolio involves the execution of multiple verification tools or configurations on the same verification problem, either in parallel or in a coordinated sequence. The main idea is to make use of the complementary strengths of different verifiers, as no single tool consistently outperforms all others across the diverse landscape of verification tasks. While portfolios can combine configurations of a single tool, usually they are used to integrate tools using fundamentally different verification approaches (e.g., bounded model checking, abstract interpretation, or interpolation), improving overall robustness and increasing the likelihood of verification success within given time or resource bounds.

In recent years, the use of portfolio-based algorithm selection and techniques has grown significantly in the field of software verification [7]. Clear evidence of this fact is that approximately a third of the entries in SV-COMP 2025 [7] used some form of algorithm selection or portfolio in their verification approach. Entries include THETA [28] or CPACHECKER [5], for example, that use select different algorithms based on the input problem, or CPV [10] that uses a portfolio of other tools for verification.

2.3 COVERITEAM

COVERITEAM [4] is a framework for constructing and executing modular verification workflows by composing existing verification tools into customizable pipelines. It builds on three basic concepts: verification artifacts, actors, and compositional operators.

1. Verification artifacts depict the data in the workflow, such as the input files, the property to verify, the results, and witnesses.
2. Actors, on the other hand, act on the data and execute tools such as verifiers, validators, or test generators. They take verification artifacts as inputs and produce verification artifacts as outputs.
3. Compositional operators provide a way to tie multiple actors together by taking care of scheduling tasks and directing the verification artifacts between the actors.

COVERITEAM is tightly integrated with the toolchain of SV-COMP. It executes the actors by using BENCHEXEC [6] under the hood for reliable containerization and measurement. Moreover, as part of the submission process to SV-COMP, tool authors must submit a tool entry file to the FM Tools repository[1] that CoVeriTeam can consume as an actor definition. The tool entry file contains, among other information:

- contains details on where to download the tool from. It supports DOIs for Zenodo archives or URLs for direct repository artifacts;
- describes which command-line parameters to pass to the tool;
- determines which BenchExec tool-info module to use that integrates the tool with the benchmark environment;
- lists dependencies that the system must have for the tool to run.

Leveraging this tight integration, CoVeriTeam is able to execute any tool from the last couple of editions of SV-COMP out-of-the-box, and makes it easy and well documented to integrate additional tools with it.

3 Verification Approach

In this section, we present our approach to the verification of CHCs. Our main idea is based on our previous work in [26, 2] in which we presented a novel way to verify CHCs by transforming them into a C program (①). This transformation is done in such a way that the reachability of an assertion failure in the C program corresponds to the satisfiability of the CHC. The C program is then verified using a software verification tool (②) that produces a correctness witness if the program is safe (assertion failure is not reachable), or a violation witness if the program is unsafe (assertion failure is reachable). Finally, the verdicts of the software verification tool are transformed into the CHC domain (③): a safe verdict implies a satisfiable CHC, and the model can be extracted from the correctness witness, while an unsafe verdict means an unsatisfiable CHC, and the refutation can be derived from the violation witness. The approach is summarized in Figure 1.

In this paper, we focus only on ②. More specifically, we aim to determine an optimal portfolio of software verification tools for C programs produced from CHCs. With this approach, we aim to diversify the field of CHC solving further, especially for theories where tool support is sparse (e.g., bitvectors).

4 Portfolio construction

This section elaborates on the main ideas and the experiments conducted to determine the optimal portfolio. First, we discuss CHCs containing linear integer arithmetic and elaborate on how we overcame the challenges posed by the C standard for this category. Then, we explain our portfolio approach and present the results supporting our portfolios for CHCs containing bitvercors.

4.1 Linear integer arithmetic

The theory of linear integer arithmetic (LIA) deals with formulas over integer variables using linear expressions and supporting operations such as addition, subtraction, and comparisons, but excludes the multiplication of variables or non-linear terms.

[1] `https://gitlab.com/sosy-lab/benchmarking/fm-tools`

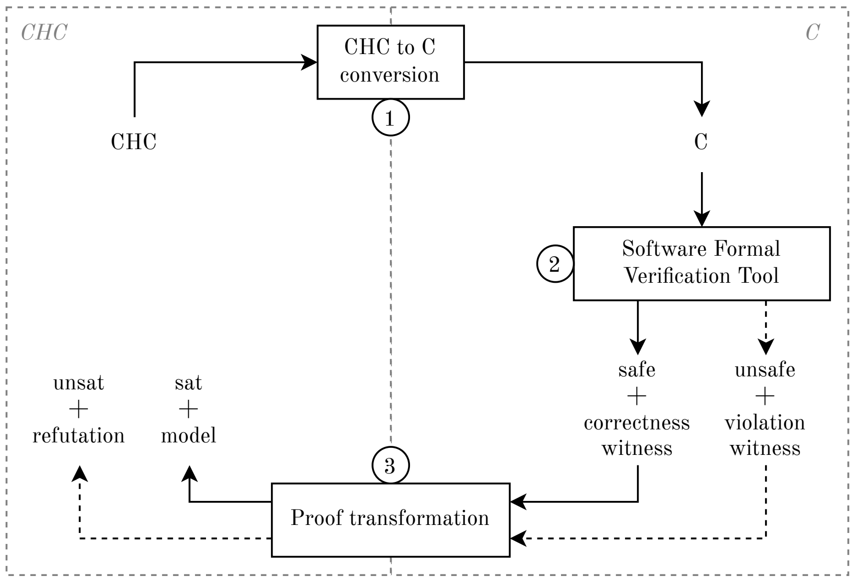

CHC
C
CHC to C conversion
CHC
C
1
2
Software Formal Verification Tool
safe + correctness witness
unsafe + violation witness
unsat + refutation
sat + model
3
Proof transformation

Figure 1: Overview of the proposed approach

One of the key differences between CHCs and C programs in handling integers is the way in which they are interpreted. CHCs use unbounded integers as they reason over the mathematical set of integers. C programs, on the other hand, are expected to run on hardware with finite resources. Thus, every value (in memory) is expressed over a finite number of bits, leading to a bounded domain.

This difference in semantics means that not all CHCs containing linear integer arithmetic can be mapped to a C program with equivalent behavior. Some CHCs might require the corresponding C program to evaluate expressions to values outside the possible domain; in other words, overflow. If the overflowing value is of an unsigned type, the value wraps around the domain, which does not match the behavior of such values in CHCs. On the other hand, if the overflowing value is of a signed type, the resulting behavior is undefined according to the C standard. Due to overflow, our approach can unfortunately be both unsound (with integer wraparounds causing infeasible traces) and incomplete (with a constrained integer domain causing incomplete models).

Fortunately, there are software verification tools that can determine whether a C program contains an overflow or not. Based on this observation, we propose the following structure for the portfolio (Figure 2). We first transform the CHC into a C program via the transformation introduced in [2]. Then, we run the reachability analysis on the C program that can produce three different verdicts:

1. If the verdict of the reachability analysis is safe, we check via an overflow analysis if the C program contains an overflow. If it does, the result is unsound, and the final verdict must be unknown. If there is no overflow in the problem, then we return a safe (satisfiable) result.
2. If the verdict of the reachability analysis is unsafe, we generate a test case from the violation witness. If the execution of the witness leads to an overflow, the result is unsound, and the final verdict is unknown. If there is no overflow, the witness corresponds to a sound refutation, and we return an unsafe (unsatisfiable) result.
3. If the verdict of the reachability analysis is unknown, we return that unknown verdict.

To determine which tools to use, we opted to run experiments. We transformed approximately 600

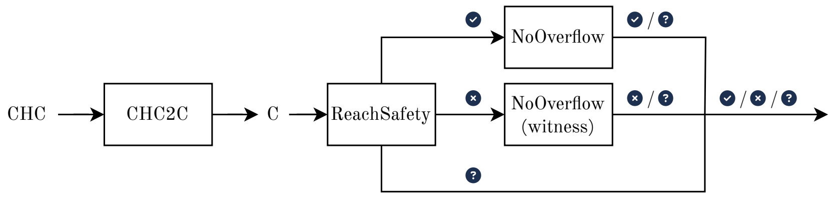


Figure 2: The proposed portfolio for the LIA theory

CHCs from the benchmark set[2] of CHC-COMP 2025 into C programs with both the recursive backward and the nonrecursive forward transformation. In the end, we ended up with 557 recursive C programs from the backward transformation of all CHCs and 195 nonrecursive C programs from the forward transformation of linear CHCs. We conducted our experiments using BenchExec [6].

We selected 16 different tools based on the results of the NoOverflow category of SV-COMP 2025 [7]. We ran each benchmark with a time limit of 15 minutes, on 4 CPU cores and a memory limit of 15 GB of RAM. The results can be seen in Table 1. The first column denotes whether the C program resulted from the nonrecursive forward transformation or the recursive backward transformation. The verdict column describes the verdict given by the overflow checker, while the verdict type describes whether the given verdict was correct or not. The rest of the columns contain the number of overflow tasks solved with the given resources.

The only tool that produced wrong results was SVF-SVC, but it produced more incorrect verdicts than correct ones, so it is less than an ideal candidate tool for verification. There were also tools (e.g., Goblint, Theta) that were able to prove if a program did not contain overflow, but failed to find an instance of overflow during the experiment, also rendering them unsuitable candidates for this task. Taking also into account the tool diversity as well (UKojak, UTaipan, and UAutomizer are the same tool family), we opted to use Bubaak, Symbiotic, UAutomizer, and ESBMC-KIND for overflow analysis.

Table 1: The number of solved overflow tasks with LIA

Recursive	**Verdict**	**Verdict type**	goblint	mopsa	emergentheta	sv-sanitizers	thorn	theta	svf-svc	bubaak-split	cpachecker	2ls	ukojak	bubaak	utaipan	symbiotic	uautomizer	esbmc-kind	**Out of**
False	No overflow	confirmed	26	24	22	9	22	22	81	21	27	28	32	21	33	21	36	0	195
	Overflow	confirmed	0	0	0	0	0	0	0	61	65	75	86	92	93	93	96	105	
		wrong	0	0	0	0	0	0	105	0	0	0	0	0	0	0	0	0	
True	No overflow	confirmed	45	28	43	43	43	43	224	28	65	72	79	29	76	66	93	88	557
	Overflow	confirmed	0	0	0	0	0	0	0	9	68	177	217	263	276	280	288	289	
		wrong	0	0	0	0	0	0	304	0	0	0	0	0	0	0	0	0	

For the reachability analysis, we selected 20 tools based on the results of the ReachSafety category of SV-COMP 2025. We ran each benchmark with a time limit of 15 minutes, on 4 CPU cores and a memory limit of 15 GB of RAM, similarly to the overflow experiments. The results can be seen in Table 2. The first column denotes whether the C program resulted from the forward or backward transformation. The

[2]`https://github.com/chc-comp/chc-comp24-benchmarks`

Table 2: The number of solved reachability tasks with LIA

Recursive	**Verdict**	**Verdict type**	2ls	aise	brick	bubaak	bubaak-split	cpachecker	cpv	emergentheta	esbmc-kind	goblint	hornix	korn	mopsa	svf-svc	symbiotic	theta	thorn	uautomizer	ukojak	utaipan	**Out of**
False	safe	confirmed	14	3	0	115	118	43	70	43	51	11	12	0	0	9	127	2	73	111	85	85	195
		unconfirmed	0	0	0	20	20	0	0	1	1	0	0	0	0	0	22	0	0	2	1	1	
		wrong	0	0	0	18	22	0	0	0	1	0	0	0	0	0	37	0	4	5	4	2	
	unsafe	confirmed	15	24	0	19	7	25	26	13	21	25	0	0	0	0	0	20	24	28	27	27	
		unconfirmed	1	2	0	2	2	4	1	2	0	2	0	0	0	0	0	2	1	0	0	0	
		wrong	13	14	0	12	9	14	6	13	0	18	0	0	0	0	0	14	0	0	0	0	
True	safe	confirmed	21	6	0	24	25	60	96	19	18	21	0	13	13	12	336	23	24	19	132	132	557
		unconfirmed	0	0	0	0	0	0	0	0	0	0	0	0	0	0	50	0	0	0	0	0	
		wrong	0	0	0	0	0	2	3	0	0	0	0	0	0	0	142	0	1	0	0	0	
	unsafe	confirmed	13	2	0	80	41	61	69	8	11	110	0	7	1	0	0	84	24	12	107	101	
		unconfirmed	0	0	0	2	2	1	0	0	0	2	0	0	0	0	0	2	1	0	0	0	
		wrong	3	0	0	20	16	16	1	3	0	24	0	0	0	0	0	20	1	0	0	0	

Table 3: The number of solved CHC problems by CHC solvers for the LIA theory

Verdict	eldarica	golem	ultimateunihorn	spacer	**Out of**
sat	286	256	149	337	557
unsat	151	164	125	160	

verdict column describes the verdict given by the reachability checker, while the verdict type describes whether the given verdict was correct or not. An unconfirmed verdict means that the correctness of the verdict is unknown, as it was neither confirmed nor refuted by any of the CHC solvers. The remaining columns contain the number of reachability tasks solved with the given resources.

The table highlights a wide variance in tool performance. For nonrecursive tasks, tools like BUBAAK [9] and BUBAAK-SPLIT [8] show strong results with many confirmed safe verdicts (115 and 118 tasks, respectively). Taking unsafe verdicts into account as well, THORN [27] and UAUTOMIZER [19] rise as suitable candidates. Conversely, for recursive tasks, tools CPV [10], UKOJAK [25], and UTAIPAN [14] lead the charge. The number of unconfirmed and wrong verdicts varies across tools and verdict types, indicating differences in precision and reliability. In the end, we opted to use THORN, BUBAAK, and UTAIPAN for the nonrecursive programs, while CPV and UKOJAK for the recursive programs.

Finally, we used CPA-WITNESS2TEST [3] to validate the unsafe result. Since converting violation witnesses to test cases and running them is not a resource-intensive task, and CPA-WITNESS2TEST managed to handle every violation witness our tools produced, we opted not to explore other tools.

In comparison, we included the result of some CHC solvers on the same benchmark set in Table 3. It can be seen that while software verification tools tend to solve around 100 CHCs, the CHC solvers are in the magnitude of 300-400.

4.2 Bitprecise arithmetic

The verification of CHCs with linear integer arithmetic had to use a portfolio of both reachability and overflow analysis tools to bridge the gap between the semantics of mathematical integers and values in a C program. While CHCs with mathematical integers are prevalent in practice in the CHC-COMP benchmark suite, some CHCs use other theories, most notably bitvectors.

Bitvectors represent the values on a finite number of bits and, in 2's complement, capture the semantics of C values better than mathematical integers can. Unlike mathematical integers, bitvectors provide bit-level access to the value and wrap around when a value is out of their bounds.

While bitvectors are widely used, tool support in CHC solvers for bitvectors is sparse. Of the tools participating in CHC-COMP 2024, only THETA [26], and ELDARICA [20] were able to verify CHC problems with bitvectors. On the other hand, more than 25 tools participated in the ReachSafety-BitVectors category on SV-COMP 2024. Making use of these tools for CHC solving would significantly diversify solving capabilities.

As bitvectors capture the semantics of C values, there is no need to check for overflow or validate the result afterwards: the safety of the program implies the satisfiability of the CHC problem directly. Therefore, it is sufficient to perform a reachability analysis only for CHC problems with bitvectors.

We transformed the CHCs in the bitvectors category of the CHC-COMP 2024 benchmark suite^3, and ended up with 213 nonrecursive C programs and 396 recursive ones. We selected seven different tools based on the results of the ReachSafety-Bitvector category of SV-COMP 2025 [7]. We ran each benchmark with a time limit of 15 minutes, on 2 CPU cores and a memory limit of 15 GB of RAM. The results of the experiments are in Table 4.

Notably, in the nonrecursive setting, most tools achieve a relatively high number of confirmed unsafe verdicts, indicating strong effectiveness in proving unsatisfiability. Tools such as CPACHECKER [1], ESBMC-KIND [30], and SYMBIOTIC [22] show consistently high confirmed unsat results. The number of confirmed safe verdicts is more modest and varies across tools (especially taking wrong verdicts into account), with CPACHECKER achieving relatively higher counts. For recursive tasks, the landscape is similar, with the notable exception that only CPACHECKER was able to produce more than a couple of safe verdicts.

We also included the performance of two CHC solvers on the same benchmark set in Table 5 for comparison. It can be seen that while the CHC solvers still perform better, the gap is far less than previously with linear integer arithmetic. Moreover, the relatively high ratio of unconfirmed to confirmed verdicts shows that the software tools solved numerous tasks that none of the CHC solvers were able to. In the end, we opted to use a portfolio of CPACHECKER, ESBMC-KIND, and SYMBIOTIC.

4.3 Final portfolio

Based on the previous findings, we opted to use the following portfolio at the end. We start by first transforming the CHC into a C program nonrecursively. After that, based on the theory used, the workflow diverges. If the CHC uses LIA, first, a reachability analysis is conducted with THORN, BUBAAK, and UTAIPAN in a parallel configuration (meaning the tools are running until one of them produces a result). Then, based on the verdict, the verdict is either validated via CPA-WITNESS2TEST or by an overflow check with BUBAAK, SYMBIOTIC, UAUTOMIZER, and ESBMC-KIND in a parallel configuration. If the CHC uses bitvectors, only a single reachability analysis is conducted by CPACHECKER, ESBMC-

^{3}`https://github.com/chc-comp/chc-comp24-benchmarks`

Table 4: The number of solved reachability tasks for bitvectors

Recursive	Verdict	Verdict type	2ls	bubaak-split	cpachecker	emergentheta	esbmc-kind	symbiotic	theta	Out of
False	safe	confirmed	9	39	27	16	5	0	38	213
		unconfirmed	4	28	6	21	4	0	32	
		wrong	0	16	0	4	0	0	8	
	unsafe	confirmed	31	62	81	69	79	75	68	
		unconfirmed	5	8	12	3	10	10	3	
True	safe	confirmed	0	0	24	0	0	0	0	396
		unconfirmed	1	1	13	5	1	1	6	
		wrong	0	0	0	2	0	0	4	
	unsafe	confirmed	69	88	91	67	101	99	73	
		unconfirmed	4	9	10	0	18	16	4	

Table 5: The number of solved CHC problems by CHC solvers for the bitvector theory

Verdict	eldarica	theta	Out of
sat	103	44	396
unsat	160	161	

KIND, and SYMBIOTIC in a parallel configuration. If the analysis produces a result, the portfolio ends its run.

If the nonrecursive reachability or overflow analysis does not yield a result under a given time limit, the nonrecursive analysis is terminated. Next, the recursive backward transformation transforms the CHC into a C program. After that, the portfolio diverges again based on the theory used by the CHC. In the case of LIA, a reachability analysis by CPV and UKOJAK is followed by an overflow analysis with BUBAAK, SYMBIOTIC, UAUTOMIZER, and ESBMC-KIND in a parallel configuration. If the CHC uses bitvectors, only a single reachability analysis is conducted by CPACHECKER, ESBMC-KIND, and SYMBIOTIC.

We implemented the proposed portfolio in the open-source CHCVERIF [15] tool using COVERITEAM and using the archives of the aforementioned tools from SV-COMP 2024.

5 Discussion

Our results demonstrate the viability and potential of using software verification tools as a backend for CHC solving, particularly when supported by a carefully constructed portfolio. This strategy opens up new directions for verifying CHCs over theories with limited or no dedicated solver support, such as bitvectors.

Bitvectors, in particular, represent a challenging theory for traditional CHC solvers due to their low-level, word-based semantics and their close correspondence with hardware behavior. However, bitvectors are widely supported by state-of-the-art software verifiers. By translating CHCs into C code, our approach effectively leverages the mature tool ecosystem of software verification to handle such cases, allowing us to shift the burden of reasoning about bit-level behavior onto tools that are already optimized for such tasks, such as CPACHECKER, ESBMC, and SYMBIOTIC.

In the case of the more established linear integer arithmetics, the overall picture is much more nuanced. First, the software verification tools performed better on the nonrecursive forward transformation that only supports linear CHCs, but are far from the performance of CHC solvers. The fact that the backward transformation performed worse is not surprising, as it introduces recursive functions into the verification process, which are considered a more difficult verification problem and are supported by fewer tools. However, the main limitation is the poor performance of the overflow tools. As these results need to be verified by an overflow check, the poor overflow analysis performance will impact the whole portfolio.

6 Conclusion

We presented CHCVERIF, a tool for solving Constrained Horn Clause (CHC) problems by translating them to C programs and leveraging existing software verification tools to check safety properties. This approach enables the reuse of mature verification infrastructures to tackle CHC benchmarks, particularly those involving bitvectors and low-level semantics. Our evaluation shows that while the method enjoys only moderate success with linear integer arithmetic due to semantic mismatches, it achieves modest success on bitvector benchmarks, demonstrating the potential of this translation-based approach for certain classes of CHC problems. In the future, we plan to extend CHCVERIF to support more theories, such as floating-point numbers that no CHC solvers support yet.

References

[1] D. Baier, D. Beyer, P.-C. Chien, M. Jankola, M. Kettl, N.-Z. Lee, T. Lemberger, M. Lingsch-Rosenfeld, M. Spiessl, H. Wachowitz & P. Wendler (2024): CPACHECKER *2.3 with Strategy Selection (Competition Contribution)*. In: *Proc. TACAS (3)*, LNCS 14572, Springer, pp. 359–364, doi:10.1007/978-3-031-57256-2_21.

[2] Levente Bajczi & Vince Molnár (2025): *Solving Constrained Horn Clauses as C Programs with CHC2C*. In Thomas Neele & Anton Wijs, editors: *Model Checking Software*, Springer Nature Switzerland, Cham, pp. 146–163, doi:10.1007/978-3-031-66149-5_8.

[3] Dirk Beyer, Matthias Dangl, Thomas Lemberger & Michael Tautschnig (2018): *Tests from Witnesses*. In Catherine Dubois & Burkhart Wolff, editors: *Tests and Proofs*, Springer International Publishing, Cham, pp. 3–23, doi:10.1007/978-3-319-92994-1_1.

[4] Dirk Beyer & Sudeep Kanav (2022): *CoVeriTeam: On-demand composition of cooperative verification systems*. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, pp. 561–579, doi:10.1007/978-3-030-99524-9_31.

[5] Dirk Beyer & M Erkan Keremoglu (2011): *CPAchecker: A tool for configurable software verification*. In: *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, Springer, pp. 184–190, doi:10.1007/978-3-642-22110-1_16.

[6] Dirk Beyer, Stefan Löwe & Philipp Wendler (2019): *Reliable benchmarking: requirements and solutions*. *International Journal on Software Tools for Technology Transfer* 21(1), pp. 1–29, doi:10.1007/s10009-017-0469-y.

[7] Dirk Beyer & Jan Strejček (2025): *Improvements in software verification and witness validation: SV-COMP 2025*. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, pp. 151–186, doi:10.1007/978-3-031-90660-2_9.

[8] M. Chalupa & C. Richter (2024): BUBAAK-SPLIT*: Split What You Cannot Verify (Competition Contribution)*. In: *Proc. TACAS (3)*, LNCS 14572, Springer, pp. 353–358, doi:10.1007/978-3-031-57256-2_20.

[9] M. Chalupa & C. Richter (2025): BUBAAK*: Dynamic Cooperative Verification (Competition Contribution)*. In: *Proc. TACAS (3)*, LNCS 15698, Springer, pp. 212–216, doi:10.1007/978-3-031-90660-2_14.

[10] P.-C. Chien & N.-Z. Lee (2024): CPV*: A Circuit-Based Program Verifier (Competition Contribution)*. In: *Proc. TACAS (3)*, LNCS 14572, Springer, pp. 365–370, doi:10.1007/978-3-031-57256-2_22.

[11] David R. Cok (2012): *The SMT-LIBv2 Language and Tools: A Tutorial*. Available at `https://smtlib.github.io/jSMTLIB/SMTLIBTutorial.pdf`.

[12] Jakub Daniel, Alessandro Cimatti, Alberto Griggio, Stefano Tonetta & Sergio Mover (2016): *Infinite-State Liveness-to-Safety via Implicit Abstraction and Well-Founded Relations*. 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. 271–291, doi:10.1007/978-3-319-41528-4_15.

[13] Emanuele De Angelis & Hari Govind V. K. (2022): *CHC-COMP 2022: Competition Report*. In Geoffrey William Hamilton, Temesghen Kahsai & Maurizio Proietti, editors: *Proceedings 9th Workshop on Horn Clauses for Verification and Synthesis and 10th International Workshop on Verification and Program Transformation, HCVS/VPT@ETAPS 2022, and 10th International Workshop on Verification and Program TransformationMunich, Germany, 3rd April 2022*, *EPTCS* 373, pp. 44–62, doi:10.4204/EPTCS.373.5.

[14] D. Dietsch, M. Heizmann, D. Klumpp, F. Schüssele & A. Podelski (2023): ULTIMATE TAIPAN *2023 (Competition Contribution)*. In: *Proc. TACAS (2)*, LNCS 13994, Springer, pp. 582–587, doi:10.1007/978-3-031-30820-8_40.

[15] Mihály Dobos-Kovács & Levente Bajczi (2025): *chc2c-svcomp: Portfolio of software verification tools for solving systems of Horn clauses.*, doi:10.5281/zenodo.15283157.

[16] Zafer Esen & Philipp Rümmer (2022): *Tricera: Verifying C Programs Using the Theory of Heaps*. In: *2022 Formal Methods in Computer-Aided Design (FMCAD)*, pp. 380–391, doi:10.34727/2022/isbn.978-3-85448-053-2_45.

[17] Grigory Fedyukovich, Arie Gurfinkel & Aarti Gupta (2019): *Lazy but Effective Functional Synthesis*. In Constantin Enea & Ruzica Piskac, editors: *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings*, *Lecture Notes in Computer Science* 11388, Springer, pp. 92–113, doi:10.1007/978-3-030-11245-5_5.

[18] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli & Jorge A Navas (2015): *The SeaHorn verification framework*. In: *International Conference on Computer Aided Verification*, Springer, pp. 343–361, doi:10.1007/978-3-319-21690-4_20.

[19] M. Heizmann, J. Christ, D. Dietsch, E. Ermis, J. Hoenicke, M. Lindenmann, A. Nutz, C. Schilling & A. Podelski (2013): ULTIMATE AUTOMIZER *with SMTInterpol (Competition Contribution)*. In: *Proc. TACAS*, LNCS 7795, Springer, pp. 641–643, doi:10.1007/978-3-642-36742-7_53.

[20] 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.

[21] Qinheping Hu, John Cyphert, Loris D'Antoni & Thomas W. Reps (2020): *Exact and approximate methods for proving unrealizability of syntax-guided synthesis problems*. In Alastair F. Donaldson & Emina Torlak, editors: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, ACM, pp. 1128–1142, doi:10.1145/3385412.3385979.

[22] M. Jonáš, K. Kumor, J. Novák, J. Sedláček, M. Trtík, L. Zaoral, P. Ayaziová & J. Strejček (2024): SYMBIOTIC 10*: Lazy Memory Initialization and Compact Symbolic Execution (Competition Contribution)*. In: *Proc. TACAS (3)*, LNCS 14572, Springer, pp. 406–411, doi:10.1007/978-3-031-57256-2_29.

[23] Jinwoo Kim, Qinheping Hu, Loris D'Antoni & Thomas W. Reps (2021): *Semantics-guided synthesis*. *Proc. ACM Program. Lang.* 5(POPL), pp. 1–32, doi:10.1145/3434311.

[24] Anvesh Komuravelli, Arie Gurfinkel & Sagar Chaki (2016): *SMT-based model checking for recursive programs*. *Form. Methods Syst. Des.* 48(3), p. 175–205, doi:10.1007/s10703-016-0249-4.

[25] A. Nutz, D. Dietsch, M. M. Mohamed & A. Podelski (2015): ULTIMATE KOJAK *with Memory Safety Checks (Competition Contribution)*. In: *Proc. TACAS*, LNCS 9035, Springer, pp. 458–460, doi:10.1007/978-3-662-46681-0_44.

[26] Márk Somorjai, Mihály Dobos-Kovács, Zsófia Ádám, Levente Bajczi & András Vörös (2024): *Bottoms up for CHCs: novel transformation of linear constrained horn clauses to software verification*. *arXiv preprint arXiv:2404.15215*, doi:10.4204/eptcs.402.11.

[27] C. Telbisz, L. Bajczi, D. Szekeres & A. Vörös (2025): THETA*: Various Approaches for Concurrent Program Verification (Competition Contribution)*. In: *Proc. TACAS (3)*, LNCS 15698, Springer, pp. 260–265, doi:10.1007/978-3-031-90660-2_22.

[28] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei & István Majzik (2017): *Theta: a Framework for Abstraction Refinement-Based Model Checking*. In Daryl Stewart & Georg Weissenbacher, editors: *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, pp. 176–179, doi:10.23919/FMCAD.2017.8102257.

[29] Scott Wesley, Maria Christakis, Jorge A. Navas, Richard J. Trefler, Valentin Wüstholz & Arie Gurfinkel (2022): *Verifying Solidity Smart Contracts via Communication Abstraction in SmartACE*. In Bernd Finkbeiner & Thomas Wies, editors: *Verification, Model Checking, and Abstract Interpretation - 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16-18, 2022, Proceedings*, *Lecture Notes in Computer Science* 13182, Springer, pp. 425–449, doi:10.1007/978-3-030-94583-1_21.

[30] T. Wu, X. Li, E. Manino, R. Menezes, M. Gadelha, S. Xiong, N. Tihanyi, P. Petoumenos & L. Cordeiro (2025): ESBMC v7.7*: Efficient Concurrent Software Verification with Scheduling, Incremental SMT and Partial Order Reduction (Competition Contribution)*. In: *Proc. TACAS (3)*, LNCS 15698, Springer, pp. 223–228, doi:10.1007/978-3-031-90660-2_16.