RAHFT: A tool for verifying Horn clauses using abstract interpretation and finite tree automata

Bishoksan Kafle[1] *, John P. Gallagher[1 2] **, and José F. Morales[2] ***

[1] Roskilde University, Denmark
[2] IMDEA Software Institute, Madrid, Spain

Abstract. We present RAHFT (Refinement of Abstraction in Horn clauses using Finite Tree automata), an *abstraction refinement* tool for verifying safety properties of programs expressed as Horn clauses. The paper describes the architecture, strength and weakness, implementation and usage aspects of the tool. RAHFT loosely combines three powerful techniques for program verification: (i) program specialisation, (ii) abstract interpretation, and (iii) trace abstraction refinement in a non-trivial way, with the aim of exploiting their strengths and mitigating their weaknesses through the complementary techniques. It is interfaced with an abstract domain, a tool for manipulating finite tree automata and various solvers for reasoning about constraints. Its modular design and customizable components allows for experimenting with new verification techniques and tools developed for Horn clauses.

1 Constrained Horn clause verification and our approach

A constrained Horn clause (CHC) is a first order predicate logic formula usually written in the form $p(X) \leftarrow \phi, p_1(X_1), ..., p_k(X_k)$ $(k \geq 0)$ using Constraint Logic Programming (CLP) syntax, where ϕ is a first order logic formula (constraint) with respect to some background theory, X_i, X are (possibly empty) tuples of distinct variables, and $p_1, \ldots, p_k, p$ are predicate symbols. There is a distinguished predicate symbol *false* which is interpreted as FALSE. Clauses with *false* head are called *integrity constraints*. A set of CHCs is called a (CLP) program.

An interpretation of a set of CHCs P is a set of *constrained facts* of the form $A \leftarrow \phi$ where A is an atom and ϕ is a formula with respect to some background theory. An interpretation that satisfies each clause is called a *model* (a *solution* in some works [6,34]). In Horn clause verification, *integrity constraints* represent the safety properties to be verified; other clauses represent the program's behaviours. The CHC verification problem is to check whether there exists a model of P.

Several verification tools have been developed for CHCs, including SeaHorn [24], QARMC [21], VeriMap [16], Convex polyhedral analyser [31], TRACER

* Funded by the EU FP7 project 318337, *ENTRA*.
** Funded by the EU FP7 project 611004, coordination and support action ICT-Energy.
*** Work partially funded by Comunidad de Madrid project S2013/ICE-2731 N-Greens Software, MINECO Projects TIN2012-39391-C04-03 (StrongSoft) and TIN2015-67522-C3-1-R (TRACES), and EU FP7-ICT-2013.3.4 project 610686 POLCA.

[29], ELDARICA [27], and Trace abstraction refinement tool [37]. They exploit either Formulation I or Formulation II for Horn clause verification.
Formulation I (deductive): P has a model if and only if $P \nvdash \mathit{false}$ (false is not derivable from P). In CLP terminology, $P \vdash A$ if and only if the query $\leftarrow A$ succeeds in P. In this formulation it is sufficient to show that the query $\leftarrow \mathit{false}$ fails finitely or infinitely. Formulation I forms the basis of the tools described in [25,37]. As the *minimal model* of P contains exactly the set of atoms that succeed [28], we have another formulation of the CHC verification problem [20].
Formulation II (model-based): P has a model if and only if $\mathit{false} \notin M[\![P]\!]$, where $M[\![P]\!]$ is the minimal model of P. In Formulation II it is sufficient to find a model $M' \supseteq M[\![P]\!]$, where $\mathit{false} \notin M'$. It forms the basis of tools based on *abstract interpretation*, *interpolation* or *predicate abstraction* [21,24,31].

The program in Figure 1(a) is a simple but challenging problem for many verification tools. $\mathtt{1}(\mathtt{X},\mathtt{Y}) \equiv \mathtt{X} \geq \mathtt{Y} \wedge \mathtt{Y} \geq \mathtt{0}$ is a model of the program, whose solution requires the discovery of the invariants $\mathtt{X} \geq \mathtt{Y}$ and $\mathtt{Y} \geq \mathtt{0}$. For example neither QARMC [21] nor SeaHorn [24] (using only the PDR engine [7]) terminates on this program. However, SeaHorn (with PDR and the abstract interpreter IKOS [8]) solves it. RAHFT solves it with the pre-processing step alone.

RAHFT exploits both of the above formulations using techniques based on *abstract interpretation* over the domain of convex polyhedra, *trace abstraction-refinement* using finite tree automata (FTAs) and *program specialisation* using *constraint specialisation* [30]. The motivations behind this combination are: (i) to benefit from a powerful and scalable technique such as *abstract interpretation* [13] for verifying properties of programs, (ii) to refine *abstract interpretation* through automata theoretic operations which offers the advantages of simplicity and generality [31] and (iii) to construct highly parametric and configurable verification tools through *program transformation* [16].

2 RAHFT architecture and interface

Figure 1(b) gives an overview of RAHFT. It compiles to a standalone command line utility that accepts a set of CHCs as input. It consists of two modules namely, *Abstraction* (green box) and *Refinement* (red box). RAHFT takes a file containing a set of CHCs P as input and returns *safe* or *unsafe* respectively if P has or does not have a model.

2.1 Abstraction

The *Abstraction* module takes a set of CHCs P as input and returns *safe*, *unsafe* or a trace representing the abstract derivation of *false* together with the set of all derivations (traces) (both represented as FTAs) used while applying *abstraction interpretation* to P. It consists of the following components:

Pre-processor (PP): Pre-processing is a model-preserving source-to-source program transformation of Horn clauses. In principle, any such transformation can be used as a pre-processor, but we use *constraint specialisation* [30]. The

```
false :- Y>X, l(X,Y).
l(X1,Y1) :- X1=X+Y,
            Y1=Y+1,
            l(X,Y).
l(X,Y) :- X=1, Y=0.
```

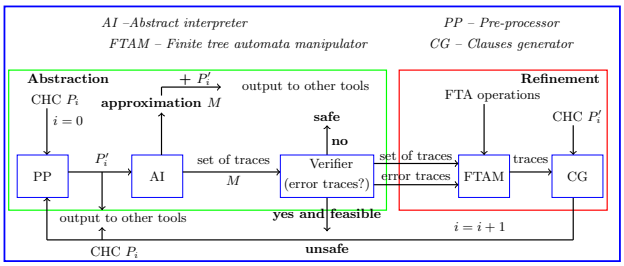

Fig. 1: (a) Example program; (b) The architecture of RAHFT.

specialisation consists of strengthening the constraints in the clauses using *abstract interpretation* [13] and *query-answer transformation* [3,17] of the original program. The specialisation is independent of the *abstract domain* and the background theory underlying the clauses and does not unfold the clauses at all. This has been proven to be an effective transformation [30] for verifying Horn clauses [15] and as a pre-processor to other Horn clause verification tools such as [21].

Abstract Interpreter (AI): The AI implements a fixed point algorithm over the domain of *convex polyhedra* [12] based on *abstract interpretation* [13]. It constructs an over-approximation M of the *minimal model* of a program P, where M contains at most one *constrained fact* $p(X) \leftarrow \phi$ for each predicate p. The constraint ϕ is a conjunction of linear inequalities, representing a convex polyhedron. The set of traces used during abstract interpretation of P can be captured by an FTA, say $\mathcal{A}_P$, using M as shown in [32]. An FTA is a mathematical model capable of capturing tree structured computations (Horn clauses derivations) (see [31] for the correspondence between a program and an FTA).

The approximation M and the pre-processed clauses can be used by other Horn clause tools, for example [21]. These tools can strengthen M (which may contain some useful invariants) incrementally to construct a model of P rather than starting from a coarse abstraction ($p(X) \leftarrow \mathit{true}$ for each predicate p of P).

Verifier: The verifier receives M and $\mathcal{A}_P$ and checks the *safety* of the clauses based on some simple condition. The clauses are safe if there is no constrained fact for *false* in M (M is called *safe inductive invariant* or a *model* of P) or there are no error traces rooted at *false*. Otherwise we do not know whether the clauses are unsafe or whether the approximation was too imprecise. In this case, the verifier picks a trace, say $t \in \mathcal{A}_P$, representing the abstract derivation of *false* (if any) from the set of traces. If t is feasible (while simulating in P), then P is unsafe and t is a *counterexample*, otherwise we refine P.

2.2 Refinement

The *Refinement* module takes as input a program P and two FTAs (i) recognising the set of all possible traces of P; and (ii) recognising a set of infeasible traces. A difference automaton is computed from these automata which recognises all traces except the infeasible ones. A refined program is obtained as output using the difference automaton and P. Rather than eliminating a single infeasible

trace in each refinement iteration, we generalise it using an *interpolant automaton* [25,32,37] thereby eliminating a possibly infinite number of infeasible traces. The refinement offers the advantages of simplicity and generality which is independent of the abstract domain and background theory underlying the clauses. The *Refinement* module consists of following components:

Finite tree automata manipulator (FTAM): FTAM takes as input two FTAs and outputs their difference automaton. The FTA difference construction needs *determinisation*; we built upon an optimised determinisation algorithm by Gallagher, Ajspur and Kafle [19] which scales well in practice, generating transitions of the determinised automaton in a very compact form called *product form*.

Clause generator (CG): Given a set of clauses P, and an automaton recognising an over-approximation of all feasible traces of P, CG produces a set of clauses which is equisatisfiable to P. For this purpose, we exploit a correspondence between the traces using the clauses and the language of FTAs to generate a new set of clauses.

The refinement offers two advantages: (i) the refinement is manifested in the clauses generated – we do not need to keep track of the previous refinements; and (ii) the original predicates get split in refined clauses which help improve the precision of analysis [20].

2.3 Implementation

RAHFT is implemented in Ciao [26] and is available from `https://github.com/bishoksan/RAHFT`. It consists of a collection of reusable Prolog modules which rely on state-of-the-art specialised external libraries written in C and C++ for handling constraints. We use the Yices SMT solver [18] and the Parma Polyhedra Library [2] for handling the constraints and the FTA library [19] for manipulating FTAs. The construction of an *interpolant tree automaton* uses an algorithm presented in [36] for computing an interpolant of two formulas. The code consists of over 7,000 lines of Ciao Prolog code split over 42 modules, interfaced to the above-mentioned external libraries. The implementation of iterative fixpoint algorithms is inspired by the approach to the abstract interpretation of logic programs described by Codish and Søndergaard [10]. Data structures for manipulating Horn clauses are based on terms and the internal Prolog database, reusing the optimizations of the underlying machine (e.g., clause indexing) rather than reimplementing them in our tool. The glue code that ties together the general purpose Prolog engine and the specialised solvers written in C and C++ is generated via the Ciao foreign interface [26].

2.4 Strength and weakness

RAHFT is a verification tool for *safety* properties of programs expressed as Horn clauses; it can be used as a *back end* solver by different *front end* tools outputting in CLP form. It handles clauses whose underlying theory is *linear arithmetic*; other theories are not supported currently. It accepts input in CLP form.

Since different components of RAHFT are loosely coupled, the tool can be reconfigured (with a very little effort) to produce verification tools solely based on (i) program transformation as in iterated specialisation approach [15] by iterating the pre-processing component, (ii) abstract interpretation, only with the AI component, (iii) trace abstraction refinement [25,37] by iterating the FTAM component, and (iv) a sensible combination thereof – all followed by a lightweight verifier which checks the *safety* of the clauses based on some condition. Since our tool uses both *state abstraction* and *trace abstraction*, it allows application of a wide range of tools and techniques.

We have evaluated RAHFT on software verification benchmarks from a variety of sources [22,29,23,4,5,27] and the results show that it compares favourably (in time and the number of instances solved) with the other state-of-the-art Horn clause verification tools (see [31,30,32] for the details).

Convex polyhedra is an expensive abstract domain and is a potential bottleneck for verification of large code bases. Instead, we can use cheaper domains supported by the Parma Polyhedra Library such as *octagons* or *intervals* at the cost of precision. RAHFT is also limited by the *hard-coded* limits of the libraries and the Prolog implementation used (e.g. *arity* limit of the predicates), which may be too restrictive for some verification problems and we intend to improve this by some suitable data representation. We are aware of some examples from SV-COMP if not many which cross this limit.

We can leverage state-of-the-art interpolating SMT solvers [9,33] for the *tree interpolant* generation which can be used for constructing an *interpolant tree automaton*; our current implementation does not scale well. Furthermore we aim to handle more advanced data structures such as arrays, maps and sets, requiring more expressive theories than linear arithmetic. One way to achieve this is by composing abstract domains as described in [14,11]; we are also aware of the support for the reduced product of domains in the PPL library.

RAHFT is able to generate a *model* (*a counterexample*) if it proves the *safety* (*unsafety*) a program. We need bookkeeping to generate these witnesses with respect to the original program; and sometimes it becomes rather challenging because of the use of external libraries, tools or the transformations applied.

3 Future work

Future work will involve making RAHFT a more flexible tool so that the user can configure other parameters such as abstract domains and pre-processors. We are also planning for a detailed performance measurement of the tool to detect bottlenecks; and work on language-based optimisations to minimize them. Generation of a *model* or a *counterexample* with respect to the original program, handling clauses with richer background theories (arrays, uninterpreted functions) is on our to-do list. In addition, we are extending RAHFT to consider Horn clauses in `SMT-LIB` format [1], though several Horn clause verification tools use standard CLP notation [31,16,21].

References

1. SMT-LIB format. `http://smtlib.cs.uiowa.edu`. Accessed: 2016-05-05.
2. R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *SCP*, 72(1-2):3–21, 2008.
3. F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the 5^{th} ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, 1986.
4. D. Beyer. Second competition on software verification - (summary of SV-COMP 2013). In Piterman and Smolka [35], pages 594–609.
5. D. Beyer. Software verification and verifiable witnesses - (report on SV-COMP 2015). In C. Baier and C. Tinelli, editors, *TACAS*, volume 9035 of *LNCS*, pages 401–416. Springer, 2015.
6. N. Bjørner, K. L. McMillan, and A. Rybalchenko. On solving universally quantified Horn clauses. In F. Logozzo and M. Fähndrich, editors, *SAS*, volume 7935 of *LNCS*, pages 105–125. Springer, 2013.
7. A. R. Bradley and Z. Manna. Property-directed incremental invariant generation. *Formal Asp. Comput.*, 20(4-5):379–405, 2008.
8. G. Brat, J. A. Navas, N. Shi, and A. Venet. IKOS: A framework for static analysis based on abstract interpretation. In D. Giannakopoulou and G. Salaün, editors, *SEFM*, volume 8702 of *LNCS*, pages 271–277. Springer, 2014.
9. A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The MathSAT5 SMT solver. In Piterman and Smolka [35], pages 93–107.
10. M. Codish and H. Søndergaard. Meta-circular abstract interpretation in Prolog. In T. Æ. Mogensen, D. A. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation, Complexity, Analysis, Transformation*, volume 2566 of *LNCS*, pages 109–134. Springer, 2002.
11. A. Cortesi, G. Costantini, and P. Ferrara. A survey on product operators in abstract interpretation. In A. Banerjee, O. Danvy, K. Doh, and J. Hatcliff, editors, *Semantics, Abstract Interpretation, and Reasoning about Programs*, volume 129 of *EPTCS*, pages 325–336, 2013.
12. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages*, pages 84–96, 1978.
13. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In R. M. Graham, M. A. Harrison, and R. Sethi, editors, *POPL*, pages 238–252. ACM, 1977.
14. P. Cousot, R. Cousot, and L. Mauborgne. The reduced product of abstract domains and the combination of decision procedures. In M. Hofmann, editor, *FOSSACS 2011*, volume 6604 of *LNCS*, pages 456–472. Springer, 2011.
15. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Program verification via iterated specialization. *SCP*, 95:149–175, 2014.
16. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verimap: A tool for verifying programs through transformations. In E. Ábrahám and K. Havelund, editors, *TACAS*, volume 8413 of *LNCS*, pages 568–574. Springer, 2014.
17. S. Debray and R. Ramakrishnan. Abstract interpretation of logic programs using magic transformations. *Journal of Logic Programming*, 18:149–176, 1994.
18. B. Dutertre. Yices 2.2. In A. Biere and R. Bloem, editors, *CAV'2014*, volume 8559 of *LNCS*, pages 737–744. Springer, July 2014.

19. J. P. Gallagher, M. Ajspur, and B. Kafle. An optimised algorithm for determinisation and completion of finite tree automata. *CoRR*, abs/1511.03595, 2015.
20. J. P. Gallagher and B. Kafle. Analysis and transformation tools for constrained Horn clause verification. *TPLP*, 14(4-5 (additional materials in online edition)):90–101, 2014.
21. S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In J. Vitek, H. Lin, and F. Tip, editors, *PLDI*, pages 405–416. ACM, 2012.
22. B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically refining abstract interpretations. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 443–458. Springer, 2008.
23. A. Gupta and A. Rybalchenko. InvGen: An efficient invariant generator. In A. Bouajjani and O. Maler, editors, *CAV*, volume 5643 of *LNCS*, pages 634–640. Springer, 2009.
24. A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The SeaHorn verification framework. In D. Kroening and C. S. Pasareanu, editors, *CAV*, volume 9206 of *LNCS*, pages 343–361. Springer, 2015.
25. M. Heizmann, J. Hoenicke, and A. Podelski. Refinement of trace abstraction. In J. Palsberg and Z. Su, editors, *SAS 2009*, volume 5673 of *LNCS*, pages 69–85. Springer, 2009.
26. M. V. Hermenegildo, F. Bueno, M. Carro, P. López-García, E. Mera, J. F. Morales, and G. Puebla. An overview of Ciao and its design philosophy. *TPLP*, 12(1-2):219–252, 2012.
27. H. Hojjat, F. Konecný, F. Garnier, R. Iosif, V. Kuncak, and P. Rümmer. A verification toolkit for numerical transition systems - tool paper. In D. Giannakopoulou and D. Méry, editors, *FM*, volume 7436 of *LCNS*, pages 247–251. Springer, 2012.
28. J. Jaffar and M. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
29. J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa. TRACER: A symbolic execution tool for verification. In P. Madhusudan and S. A. Seshia, editors, *CAV*, volume 7358 of *LNCS*, pages 758–766. Springer, 2012.
30. B. Kafle and J. P. Gallagher. Constraint specialisation in Horn clause verification. In K. Asai and K. Sagonas, editors, *Proceedings Workshop on PEPM, PEPM, Mumbai, India, January 15-17, 2015*, pages 85–90. ACM, 2015.
31. B. Kafle and J. P. Gallagher. Horn clause verification with convex polyhedral abstraction and tree automata-based refinement. *Computer Languages, Systems & Structures*, 2015.
32. B. Kafle and J. P. Gallagher. Interpolant tree automata and their application in Horn clause verification. *CoRR*, abs/1601.06521, 2016.
33. K. L. McMillan. Interpolants from Z3 proofs. In P. Bjesse and A. Slobodová, editors, *FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, pages 19–27. FMCAD Inc., 2011.
34. K. L. McMillan and A. Rybalchenko. Solving constrained Horn clauses using interpolation. Technical report, Microsoft Research, 2013.
35. N. Piterman and S. A. Smolka, editors. *TACAS*, volume 7795 of *LNCS*. Springer, 2013.
36. A. Rybalchenko and V. Sofronie-Stokkermans. Constraint solving for interpolation. *J. Symb. Comput.*, 45(11):1212–1233, 2010.
37. W. Wang and L. Jiao. Trace abstraction refinement for solving Horn clauses. Technical Report ISCAS-SKLCS-15-19, SCAS-SKLCS (Available from: http://lcs.ios.ac.cn/ wangwf/TechReportISCAS-SKLCS-15-19.pdf), Dec. 2015.