

OptiRica: Towards an Efficient Optimizing Horn Solver

Hossein Hojjat

Tehran Institute for Advanced Studies, Khatam University
University of Tehran, Iran
hojjat@ut.ac.ir

Philipp Rümmer

University of Regensburg, Germany
Uppsala University, Sweden
philipp.ruemmer@it.uu.se

This paper describes an ongoing effort to develop an optimizing version of the Eldarica Horn solver. The work starts from the observation that many kinds of optimization problems, and in particular the MaxSAT/SMT problem, can be seen as search problems on lattices. The paper presents a Scala library providing a domain-specific language (DSL) to uniformly model optimization problems of this kind, by defining, manipulating, and systematically exploring lattices with associated objective functions. The framework can be instantiated to obtain an optimizing Horn solver. As an illustration, the application of an optimizing solver for repairing software-defined networks is described.

1 Introduction

Constrained Horn Clauses have proven to sit at a sweet spot: the language of Horn clauses is expressive enough to capture interesting properties of complex systems; at the same time, their mathematical properties (in particular the existence of least models) enable efficient algorithms and solvers that scale to real-world applications. Nevertheless, for a long time, people have tried to extend the language of Horn clauses, while keeping some of its convenient properties. Among others, it was proposed to extend Horn clauses with well-foundedness predicates [5], universally quantified literals in the clause body [6], existentially quantified literals in the clause head [4], and disjunctions in heads [3].

This paper presents ongoing work in a similar direction: the development of optimizing solvers in which side conditions are formulated in terms of Horn clauses, and optimization objectives are characterized using finite lattices. This setting naturally generalizes MaxSAT/SMT [2] (and minimal unsatisfiability) to the setting of Horn clauses, and thus captures many forms of analysis and reasoning tasks; for instance, the exploration of all counterexamples of a set of clauses, the inference of safe parameters or sufficient pre-conditions, or program repair.

Our work builds on a lattice-based optimization framework designed for the purpose of interpolant exploration [9] and network repair [8]. In those earlier papers, only a prototypical implementation was provided that was not directly reusable in other contexts. This paper recapitulates the optimization framework, gives an overview of ongoing implementation work in the context of the Horn solver Eldarica [7], and illustrates the use of optimization for repair. We include several code examples showing how our new optimization library can be used to formulate and solve optimization problems.

Related work. In addition to the research already mentioned, our work is related to the computation of maximum specifications of functions [1], and weakest solutions of Horn clauses [10]. Other approaches proceed in a counterexample-guided manner, and refine solutions until a maximum or weakest solution has been found [1, 10]. The methodology behind our framework is different, as we require an upfront specification of the optimization objective and search space in the form of a finite lattice; our work is closer in spirit to MaxSAT/SMT [2].

2 The Lattice-Based Optimization Framework

2.1 The Framework

The lattice-based optimization framework [8, 9] is inspired by the MaxSAT/SMT paradigm [2], but generalizes MaxSAT in two ways: (i) where MaxSAT can be seen as a search on the powerset lattice induced by some set of constraints, we consider arbitrary finite lattices; and (ii) where side conditions in MaxSAT are given in terms of SAT or SMT constraints, the feasibility of solutions in the lattice optimization framework can be defined by any computable function.

The central definition needed to explain the framework is the notion of an *optimization lattice*; the following definition is a slightly generalized version of the definition in earlier papers [8]:

Definition 1 (Optimization lattice) *An optimization lattice is a triple $(\langle L, \sqsubseteq_L \rangle, F, \text{obj})$ consisting of a complete lattice $\langle L, \sqsubseteq_L \rangle$, a downward-closed feasibility predicate $F \subseteq L$, and a monotonically increasing objective function $\text{obj} : L \rightarrow D$ to a set D that is totally ordered.*

We call the elements in F *feasible*. Note that the predecessors of feasible elements are also feasible, and the successors of infeasible elements are also infeasible. An element $l \in L$ is *maximal feasible* if l is feasible, but all of its successors are infeasible.

Definition 2 *An element $l_{\max} \in L$ is called optimal if it is maximal feasible, and it is the case that $\text{obj}(l_{\max}) = \max\{\text{obj}(l) \mid l \in F\}$.*

Examples of useful optimization lattices are powerset lattices, interval lattices, as well as lattice products [8]. Algorithms to compute maximal feasible and optimal elements are able to handle large finite lattices (e.g., lattices with $> 10^{10}$ elements) by representing those lattices symbolically [8, 9]. Those algorithms are based on three main principles: greedy optimization, which is achieved by walking upward in a lattice until reaching a maximal feasible element; the computation of incomparable elements, to identify the next starting point in the lattice after discovering one maximal feasible element; and the inference of upper bounds on feasible elements in the lattice for early pruning. The computation of incomparable elements is related to hitting set methods used in MaxSAT [2], while upper feasibility bounds have similarities with the use of blocking clauses in SAT.

2.2 Implementation

We are in the process of developing a Scala library for lattice-based optimization.¹ At the moment, our library does not include a parser for a modeling language for expressing optimization problems; instead, we define an embedded domain-specific language (EDSL) using a set of basic lattice types, and operators to derive new lattices from those basic lattices.

An overview of the lattice classes is provided in Figure 1. In general, a lattice is symbolically described through a type `LatticeObj` of the nodes in the lattice, a partial order `latticeOrder` on those nodes, and methods for computing joins, meets, successors, and predecessors of nodes. Moreover, lattices are labeled, with a method `getLabel` mapping lattice nodes to elements of some defined label type. While `Label` is a type parameter of the `Lattice` trait, the type `LatticeObj` is an abstract type member, and should be seen as an existential type: every lattice is associated with some type of lattice nodes, but given a lattice no assumptions can in general be made about the node type.

Optimization lattices are derived from general lattices, but in addition provide an objective function `toScore` for mapping lattice nodes to some `Score` type, a feasibility predicate `isFeasible`, as

¹<https://github.com/uuverifiers/lattice-optimiser>

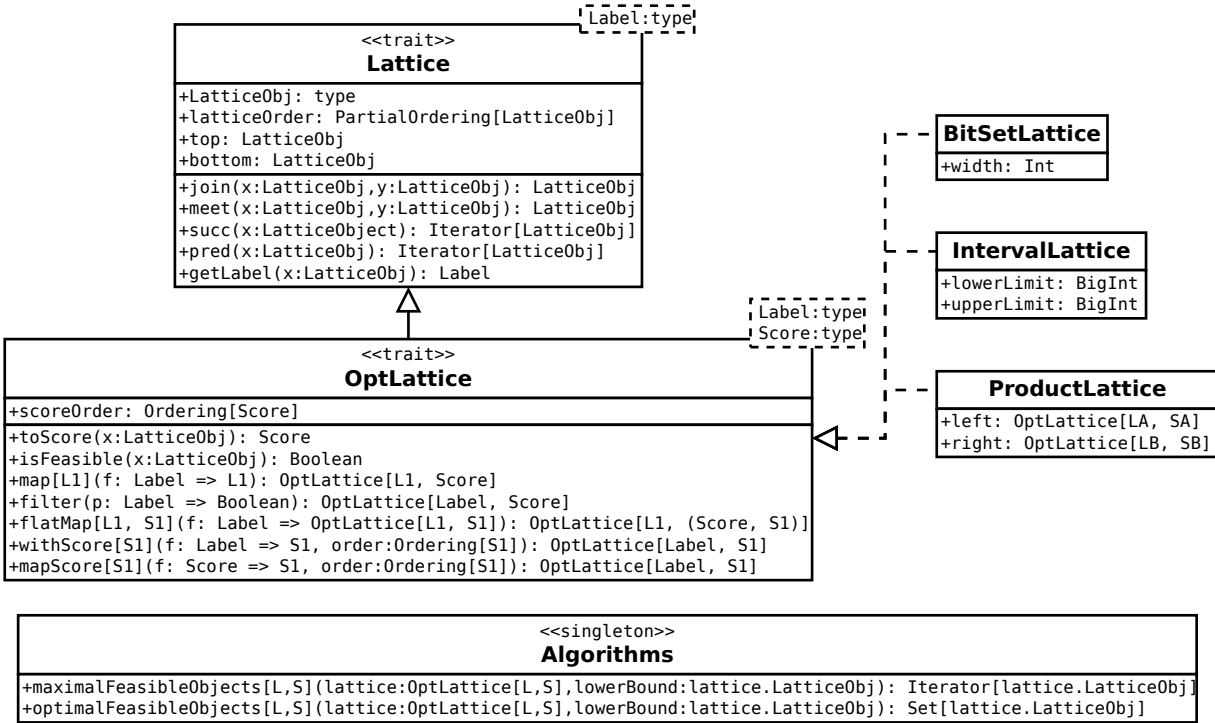


Figure 1: Class diagram of lattice types

well as a monadic interface consisting of `map`, `filter`, and `flatMap` methods. The `map` method is used to redefine the labeling of a lattice, by specifying a function from the old to a new label type; `filter` strengthens the feasibility predicate of a lattice by conjoining a new constraint; and `flatMap` is used to construct product lattices by mapping labels to optimization lattices. The objective function `toScore` can be redefined using two methods: the method `withScore` creates lattices in which the score of a node is computed as a function of the node label; and `mapScore` mutates the score of each node by applying some function `f` to it.

The functions to compute optimal elements in lattices are collected in the class **Algorithms**. The method `maximalFeasibleObjects` enumerates the maximal feasible nodes above some `lowerBound` in a given lattice; method `optimalFeasibleObjects` returns the set of maximal feasible nodes above `lowerBound` with maximum score.

Example 1 A simple example of an optimization problem expressed using the library is given in Listing 1. The code snippet² solves the problem of computing the subset of the numbers $\{0, 1, \dots, 16\}$ that (i) has the property that it does not contain any number x as well as its square x^2 ; and (ii) has the maximum element sum. To model the problem, in line 4 the powerset lattice of the set $\{0, 1, \dots, 16\}$ is declared; powerset lattices are an instance of the **BitsetLattice** class. The objective function is defined in line 5 to be the sum of the elements of a set, and the feasibility condition is defined in line 6. The sets of maximal feasible and optimal solutions are computed in lines 9 and 11, respectively. Since the methods to compute solutions are randomized, in line 0 the used random number generator is initialized.

²Complete working example in <https://github.com/uuverifiers/lattice-optimiser/blob/master/src/test/scala/lattopt/SquareTests.scala>

```

0 implicit val randomData = new SeededRandomDataSource(123)
1
2 // The powerset of the set {0, ..., 16}; subsets that contain
3 // the square of any of their elements are infeasible
4 val latt = PowerSetLattice(0 to 16).
5           withScore(_.sum).
6           filter { s => !(s exists { x => s contains x*x }) }
7
8 // there are four maximal feasible objects
9 println(Algorithms.maximalFeasibleObjects(latt)(latt.bottom).size)
10
11 // and the optimum, maximizing the sum of its elements:
12 // {2, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}
13 println(Algorithms.optimalFeasibleObjects(latt)(latt.bottom))

```

Listing 1: An optimization problem over a powerset lattice.

3 Lattice-Based Optimization Modulo Horn Constraints

3.1 Parameterized Clauses

The optimization framework can immediately be used to derive solvers for *MaxCHC*, the maximum satisfiability problem over Horn clauses. Given a set HC of constrained Horn clauses, consider the powerset lattice $\mathcal{P}(HC)$, as well as the feasibility predicate $F = \{S \subseteq HC \mid S \text{ is satisfiable}\}$ that can be implemented using an existing Horn solver. Appropriate timeouts have to be used when checking the satisfiability of subsets of HC . Solutions of MaxCHC are then the maximal feasible elements of the induced optimization lattice. Partial and weighted MaxCHC (including both hard and soft constraints, or giving weights to clauses, respectively) can be modeled by choosing an appropriate objective function *obj*.

To compute minimally unsatisfiable sets (MUSes), we can construct the inverted (or dual) powerset lattice induced by some set of clauses, and the feasibility predicate $F = \{S \subseteq HC \mid S \text{ is unsatisfiable}\}$.

The challenge, of course, is to make such constructions efficient, since the exploration of the optimization lattice will require many expensive satisfiability checks on subsets of HC . A practical implementation therefore requires an underlying *incremental* Horn solver that is tailored to the case of solving many similar queries. We are in the process of extending our solver Eldarica [7] for this purpose, using the following notion of parameterized clauses:

Definition 3 (Parameterized Clause Set) A parameterized clause is a constrained Horn clause over a set $R = R_{\text{sym}} \cup R_{\text{par}}$ of relation symbols, including a subset R_{par} of relation symbols called parameters. If $m : R_{\text{par}} \rightarrow \text{Constr}$ is a function mapping every n -ary parameter to a formula/constraint over n free variables, then the instance $HC[m]$ of a parameterized clause set HC is obtained by substituting every parameter $p \in R_{\text{par}}$ with the constraint $m(p)$.

The incremental version of Eldarica is able to directly process parameterized clauses, and postpone the instantiation of parameters as late as possible, thus minimizing the amount of work that has to be repeated when solving different clause set instances. Eldarica can also reuse counterexamples and solutions. As a future extension, we plan to add functionality to reapply CEGAR predicates across instances.

This yields the following recipe for defining optimization problems modulo Horn clauses: (i) define a set HC of parameterized Horn clauses, such that the instances of the clauses cover the intended search

```

0 import [...]
1 SimpleAPI.withProver { p =>
2   import p._
3
4   val Inv    = createRelation("I", Seq(Integer, Integer, Integer))
5   val Flag   = for (i <- 0 to 3) yield createRelation("f" + i, Seq())
6
7   val Seq(x, y, n) = createConstants(3)
8
9   val clauses = List(
10     Inv(0, 0, n)      :- (n > 0, Flag(0)()),
11     Inv(x + 1, y + 1, n) :- (Inv(x, y, n), y < n, Flag(1)()),
12     Inv(x + 2, y + 1, n) :- (Inv(x, y, n), y < n, Flag(2)()),
13     false             :- (Inv(x, y, n), x >= 2*n, Flag(3)())
14   )
15
16   def set2map(s : Set[Predicate]) =
17     (for (f <- Flag) yield (f -> if (s(f)) TRUE else FALSE)).toMap
18
19   val l1 = PowerSetLattice(Flag).withScore([...]).map(set2map(_))
20   val l2 = ClauseSatLattice(l1, clauses, Flag.toSet)
21
22   println(Algorithms.optimalFeasibleObjects(l2)(l2.bottom))
23 }

```

Listing 2: Definition of parameterized clauses, and the resulting optimization problem.

space; (ii) define a lattice $\langle L, \sqsubseteq_L \rangle$ representing the search space; each element $o \in L$ is labelled with a substitution m_o ; (iii) use one of the feasibility predicates $F_{sat} = \{o \in L \mid HC[m_o] \text{ is satisfiable}\}$ or $F_{unsat} = \{o \in L \mid HC[m_o] \text{ is unsatisfiable}\}$, implemented using an incremental Horn solver; (iv) choose a suitable objective function on the lattice. The downward-closedness of the feasibility predicate can be ensured syntactically, for instance by choosing a monotonic labeling function m_o on the lattice, and restricting parameters R_{par} to the clause bodies.

3.2 Implementation

We are developing the integration of the lattice optimization library with Eldarica as a separate library OptiRica.³ Parameterized clauses can in this setting either be created programmatically, or be read from an SMT-LIB or Prolog file using the front-end of Eldarica. In Listing 2,⁴ the former approach is chosen, and four clauses are defined over the relation symbols $R_{sym} = \{\text{Inv}\}$ and parameters $R_{par} = \{\text{Flag}(0), \dots, \text{Flag}(3)\}$ (lines 4–14). Each of the four clauses is labeled with a parameter $\text{Flag}(i)$, which can be set to *false* to disable some of the clauses.

The system consisting of all four clauses is unsatisfiable. We can therefore solve a MaxCHC problem,

³<https://github.com/uuverifiers/optirica>

⁴Complete working example in <https://github.com/uuverifiers/optirica/blob/main/src/test/scala/optirica/ClauseSatTest.scala>

and compute maximum satisfiable subsets of the four clauses. The optimization lattice for this problem has to provide the substitutions m_o instantiating parameters with formulas. For this, in line 19 the power-set lattice of the set of flags is created, equipped with some suitable objective function, and then labeled with the functions computed by `set2map`. In line 20, the class `ClauseSatLattice` is used to define those lattice nodes as feasible for which the set of instantiated clauses is satisfiable. `ClauseSatLattice` receives as arguments the lattice `l1` to be filtered, the parameterized clauses `clauses`, and the set `Flag` of parameters.

Line 20 has the same effect as a direct call to the lattice `filter` function,

```
val l2 = l1 filter { m => isSat(clauses[m]) },
```

but `ClauseSatLattice` includes the optimizations discussed above, such as applying the Horn solver Eldarica incrementally, caching results, and reusing solutions and counterexamples. The constructed lattice `l2` has three maximal feasible elements, corresponding to disabling the first, third, or fourth clause.

The OptiRica library also provides a class `ClauseUnsatLattice`, which declares those lattice nodes as feasible for which some instantiated set of clauses is unsatisfiable. In combination with an inverted powerset lattice (`PowerSet.inverted(...)`), this functionality can be used to compute MUSes.

4 A Case Study: Repair in Software-defined Networking

4.1 Overview

We have used an earlier, tailor-made implementation of an optimizing Horn solver in the domain of repairing SDN configurations [8], and outline now how this case study can be mapped to our generic framework. Consider Figure 2, which shows a three-layer topology in data centers. The network sends the packets originating from a host upward and then back downward to the destination host. Assume the host H_1 sends traffic to H_2 and H_3 , but this traffic should not reach H_4 . To implement this policy, the operator installs, e.g., a forwarding rule at C_1 to filter packets from H_1 going towards A_4 and also disable the link A_3-T_4 for good measure (in the figure, this disabled link is indicated by a “+” symbol.)

For maintenance, the network operator has turned off the core switch C_2 . When the network operator brings back C_2 , it causes a safety violation, since there is a new path from H_1 to H_4 . There are multiple repair solutions to this violation: the repair engine may disconnect the links A_1-C_2 and A_2-C_2 , or take

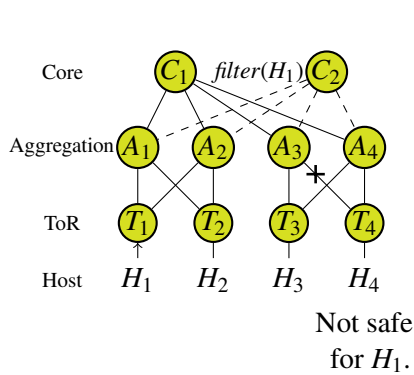


Figure 2: Software-defined network [8]

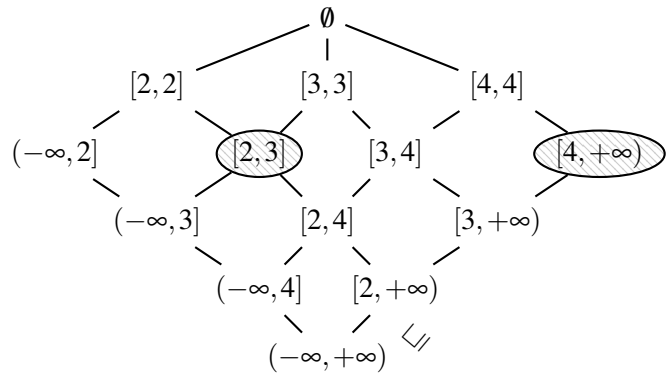


Figure 3: Inverted interval lattice [8]

```

0 val dstFilter = createRelation("dstFilter", Seq(Sort.Integer))
1 val typFilter = createRelation("typFilter", Seq(Sort.Integer))
2
3 val clauses = List(
4   [...]
5   t4(dst,typ) :- (a4(dst,typ), dst == 4,
6                  dstFilter(dst), typFilter(typ))),
7   [...]
8 )
9
10 // Possible dst filters. v(0) refers to the argument of dstFilter
11 val dstLatt = for (s <- PowerSetLattice(1 to 4))
12   yield disjFor(for (t <- s) yield v(0) == t)
13
14 // Possible typ filters. v(0) refers to the argument of typFilter
15 val typLatt = for (s <- PowerSetLattice(0 to 7))
16   yield disjFor(for (t <- s) yield v(0) == t)
17
18 val latt1 = for (c1 <- dstLatt; c2 <- typLatt)
19   yield Map(dstFilter -> c1, typFilter -> c2)
20 val latt2 = latt1 mapScore { p => p._1 + p._2 }
21 val latt3 = ClauseSatLattice(latt2, clauses,
22   Set(dstFilter, typFilter))
23
24 println(Algorithms.optimalFeasibleObjects(latt3)(latt3.bottom))

```

Listing 3: Derivation of destination filters `dstFilter` and type filters `typFilter` for the link A_4-T_4 .

C_2 offline and return the network to its initial state, or rewrite the traffic from H_1 to another type of traffic by modifying packet headers, or add filters for H_1 traffic on a number of links: $\{A_1-C_2, A_2-C_2\}$, $\{C_2-A_4, A_4-T_4\}$, $\{A_4-T_4\}$, etc.

To derive such possible repair strategies, we assume that the safety of the network has been modelled as a set of Horn clauses [8]. This set includes clauses representing the topology and the configuration of the network, for instance will the sending of packets from A_4 to T_4 be modelled as a clause $s_{T_4}(pkt', trc') \leftarrow s_{A_4}(pkt, trc) \wedge \phi$. Intuitively, a predicate $s_{T_4}(pkt, trc)$ expresses that packets pkt can reach T_4 via path trc . To search for repairs that involve filtering between A_4 and T_4 , the clause can be replaced with a parameterized clause $s_{T_4}(pkt', trc') \leftarrow s_{A_4}(pkt, trc) \wedge filter(pkt) \wedge \phi$. The relation symbol $filter \in R_{par}$ is the parameter, and will be substituted with concrete constraints during optimization. To search for repairs that will block packets to a certain port range, an interval lattice like in Figure 3 can be chosen, and the substitutions be defined as $m_{[l,u]}(filter) = (pkt.port \notin [l, u])$. Maximal feasible elements in the lattice represent safe configurations in which a minimal range of ports is blocked; for instance, the intervals $(2, 3)$ and $[4, +\infty)$. To model a larger space of possible repairs, multiple parameters can be introduced, and a product optimization lattice be chosen. To express the preference of certain kinds of repairs, an appropriate objective function can be added.

4.2 Implementation

A simple model of the network⁵ includes four clauses defining network ingress, 40 clauses for the network, and property clauses (e.g., H_1 traffic not reaching H_4). This model enables us to compute different kinds of repairs to eliminate the path from H_1 to H_4 . Using a MaxCHC encoding, as in Section 3.2, it can be derived that there are 12 different minimal sets of network clauses (i.e., maximal feasible lattice nodes) that can be removed to re-establish a safe network; two of those only require to disable a single link, namely either $A_4 - T_4$ or $C_2 - A_4$.

The derivation of more fine-grained filters is illustrated in Listing 3. We show a simplified setting in which predicates only consider the destination and type of packets; type 0 is associated with packets originating from H_1 . To infer filters for the link $A_4 - T_4$, we rewrite the corresponding clause to include predicates `dstFilter` and `typFilter` in its body (lines 5–6), and define lattices describing the possible choices of filtering. In lines 11–12, a destination filter is defined by choosing a subset s of the set $\{1, \dots, 4\}$, and then constructing the formula $\bigvee_{i \in s} x = i$. In lines 15–16, similarly filters on type are defined. Lines 18–19 create the product of the two lattices and define the resulting substitution. The objective function is defined to be the total number of destinations and types that are not filtered out in line 20. This optimization problem has two optimal solutions, namely to filter out either packets of type 0 or packets with destination H_4 on link $A_4 - T_4$.

5 Conclusions

We have outlined work towards optimizing Horn solvers. While the presented research is still work in progress, we believe that the lattice-optimization libraries are already useful tools at this point. Next steps in this project include the implementation of further types of lattices, an improved, more efficient interface between the library and the Horn solver, and a more polished language (e.g., SMT-LIB-based) for making optimization available to users.

Acknowledgments. We would like to thank the reviewers for helpful comments. Philipp Rümmer is supported by the Swedish Research Council (VR) under grant 2018-04727, by the Swedish Foundation for Strategic Research (SSF) under the project WebSec (Ref. RIT17-0011), by the Wallenberg project UPDATE, and by grants from Microsoft and Amazon Web Services.

References

- [1] Aws Albarghouthi, Isil Dillig & Arie Gurfinkel (2016): *Maximal specification synthesis*. In Rastislav Bodík & Rupak Majumdar, editors: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, ACM, pp. 789–801, doi:10.1145/2837614.2837628.
- [2] Fahiem Bacchus, Matti Järvisalo & Ruben Martins (2021): *Maximum Satisfiability*. In Armin Biere, Marijn Heule, Hans van Maaren & Toby Walsh, editors: *Handbook of Satisfiability*, 2 edition, Frontiers in Artificial Intelligence and Applications, IOS PRESS, Netherlands, pp. 929 – 991, doi:10.3233/FAIA201008.
- [3] Tewodros A. Beyene (2015): *Temporal Program Verification and Synthesis as Horn Constraints Solving*. Ph.D. thesis, TU Munich.

⁵<https://github.com/uuverifiers/optirica/blob/main/src/test/scala/optirica/NetworkTest.scala>

- [4] Tewodros A. Beyene, Corneliu Popeea & Andrey Rybalchenko (2013): *Solving Existentially Quantified Horn Clauses*. In Natasha Sharygina & Helmut Veith, editors: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings, Lecture Notes in Computer Science* 8044, Springer, pp. 869–882, doi:10.1007/978-3-642-39799-8_61.
- [5] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan & Andrey Rybalchenko (2015): *Horn Clause Solvers for Program Verification*. In Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner & Wolfram Schulte, editors: *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday, Lecture Notes in Computer Science* 9300, Springer, pp. 24–51, doi:10.1007/978-3-319-23534-9_2.
- [6] Nikolaj Bjørner, Kenneth L. McMillan & Andrey Rybalchenko (2013): *On Solving Universally Quantified Horn Clauses*. In Francesco Logozzo & Manuel Fähndrich, editors: *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings, Lecture Notes in Computer Science* 7935, Springer, pp. 105–125, doi:10.1007/978-3-642-38856-9_8.
- [7] Hossein Hojjat & Philipp Rümmer (2018): *The ELDARICA Horn Solver*. In Nikolaj Bjørner & Arie Gurfinkel, editors: *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018, IEEE*, pp. 1–7, doi:10.23919/FMCAD.2018.8603013.
- [8] Hossein Hojjat, Philipp Rümmer, Jedidiah McClurg, Pavol Cerný & Nate Foster (2016): *Optimizing Horn solvers for network repair*. In Ruzica Piskac & Muralidhar Talupur, editors: *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016, IEEE*, pp. 73–80, doi:10.1109/FMCAD.2016.7886663.
- [9] Jérôme Leroux, Philipp Rümmer & Pavle Subotic (2016): *Guiding Craig interpolation with domain-specific abstractions*. *Acta Informatica* 53(4), pp. 387–424, doi:10.1007/s00236-015-0236-z.
- [10] Sumanth Prabhu S, Grigory Fedyukovich, Kumar Madhukar & Deepak D’Souza (2021): *Specification synthesis with constrained Horn clauses*. In Stephen N. Freund & Eran Yahav, editors: *PLDI ’21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021, ACM*, pp. 1203–1217, doi:10.1145/3453483.3454104.