The ELDARICA Horn Solver

Hossein Hojjat
Rochester Institute of Technology, University of Tehran
hh@cs.rit.edu

Philipp Rümmer
Uppsala University
philipp.ruemmer@it.uu.se

***Abstract*—This paper presents the ELDARICA version 2 model checker. Over the last years we have been developing and maintaining ELDARICA as a state-of-the-art solver for Horn clauses over integer arithmetic. In the version 2, we have extended the solver to support also algebraic data types and bit-vectors, theories that are commonly applied in verification, but currently unsupported by most Horn solvers. This paper describes the high-level structure of the tool and the interface that it provides to other applications. We also report on an evaluation of the tool. While some of the techniques in ELDARICA have been documented in research papers over the last years, this is the first tool paper describing ELDARICA in its entirety.**

I. INTRODUCTION

In recent years, the computer-aided verification community has been advocating Horn clause solving as a uniform framework for reasoning about different aspects of software safety [7], [20], [32], [25]. Horn clauses form a fragment of first-order logic, modulo various background theories, in which models can be constructed effectively with the help of model checking algorithms. Horn clauses can be used as an intermediate verification language that elegantly captures various classes of systems (e.g., sequential code, programs with functions and procedures, concurrent programs, or networks of timed automata) and various verification methodologies (e.g., the use of state invariants, verification with the help of contracts, Owicki-Gries-style invariants, or rely-guarantee methods). Horn solvers can be used as *off-the-shelf backends* in verifiers, and thus enable construction of verification systems in a modular way.

ELDARICA first appeared as a solver for Horn clauses over Presburger arithmetic in 2013 [32].[1] It combines Predicate Abstraction [19] with Counterexample-Guided Abstraction Refinement (CEGAR) [12] to automatically check whether a given set of Horn clauses is satisfiable. The tool has been significantly improved since then and can now solve problems over the theories of integers, algebraic data-types [24], and bit-vectors. It can process Horn clauses and programs in a variety of formats, implements sophisticated algorithms to solve tricky systems of clauses without diverging, and offers an elegant API for programmatic use.

A. An Initial Example

To verify systems using Horn clauses, we first need to fix a set R of uninterpreted fixed-arity relation symbols, which represent the *unknowns* in the Horn clauses. A *constrained Horn clause* is a formula $H \leftarrow C \wedge B_1 \wedge \cdots \wedge B_n$ where

- C is a constraint over some background theory;
- each B_i is an application $p(t_1, \ldots, t_k)$ of a relation symbol $p \in R$ to first-order terms, usually including first-order variables;
- H is similarly either an application $p(t_1, \ldots, t_k)$ of $p \in R$ to first-order terms, or false.

H is called the *head* of the clause, $C \wedge B_1 \wedge \cdots \wedge B_n$ the *body*. In case $C = \mathit{true}$, we usually leave out C and just write $H \leftarrow B_1 \wedge \cdots \wedge B_n$. First-order variables in a clause are implicitly universally quantified; relation symbols represent set-theoretic relations over the universe U of a structure $(U, I) \in S$.

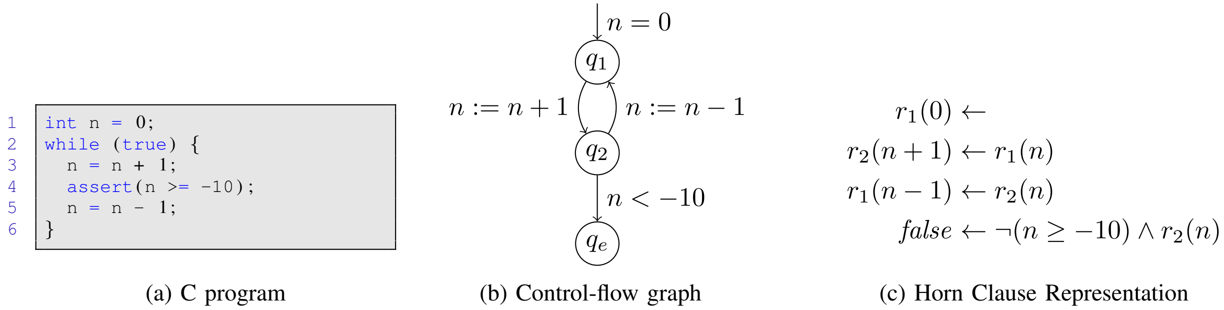
A solution to a set of Horn clauses assigns a formula to each relation symbol in such a way that all Horn clauses become valid formulas, considering first-order variables as implicitly universally quantified. When no solution exists, a derivation of *false* can be constructed as a counterexample.

Figure 1 shows a simple C program, together with a control-flow graph illustrating the program structure. The verification task consists of proving that the assertion in the program can never fail, i.e., showing program safety. In order to extract a set of Horn clauses that encode program safety, relation symbols $R = \{r_1, r_2\}$ representing state invariants of the program are introduced. The arguments of the relation symbols correspond to the values of program variables that are in scope at a particular location; in this case, to the value of n. The Horn clauses in Figure 1c represent the program transitions, and include a clause with empty body for the function entry point, two clauses corresponding to the assignments in the body of the loop, and an assertion clause with head false for the program assertion.

The clauses are constructed in such a way that safety of the program is equivalent to satisfiability of the Horn clauses. Solvers search for solutions of the Horn clauses with the help of techniques like CEGAR (e.g., in HSF [20] or ELDARICA) or IC3/PDR (e.g., in Z3 [21]). Beyond just sequential programs, Horn clauses can elegantly represent also concurrent programs, programs with functions and procedures, or timed and parameterized systems (e.g., [20], [25]).

In a verification system based on Horn clauses, Horn solvers are typically interfaced either using a textual format, most often just a Horn dialect of SMT-LIB [6], or programmatically. Figure 2 shows the Horn clauses from Figure 1 in SMT-LIB, assuming that the program variable n ranges over mathematical integers. The corresponding clauses in signed bit-vector

[1] https://github.com/uuverifiers/eldarica/



(a) C program (b) Control-flow graph (c) Horn Clause Representation

Fig. 1: Sample code with corresponding Control-Flow-Graph and Horn clauses.
The clauses are satisfied by setting $r_1(n) \equiv (n = 0)$ and $r_2(n) \equiv (n = 1)$.

```
1  (set-logic HORN)
2
3  (declare-fun r1 (Int) Bool)
4  (declare-fun r2 (Int) Bool)
5
6  (assert (r1 0))
7  (assert (forall ((n Int))
8           (=> (r1 n) (r2 (+ n 1)))))
9  (assert (forall ((n Int))
10          (=> (r2 n) (r1 (- n 1)))))
11 (assert (forall ((n Int))
12          (=> (and (r2 n) (not (>= n (- 10)))) false)))
13
14 (check-sat)
```

Fig. 2: The example from Figure 1 in SMT-LIB notation, with mathematical integer semantics.

```
1  (set-logic HORN)
2
3  (declare-fun r1 ((_ BitVec 32)) Bool)
4  (declare-fun r2 ((_ BitVec 32)) Bool)
5
6  (assert (r1 (_ bv0 32)))
7  (assert (forall ((n (_ BitVec 32)))
8           (=> (r1 n) (r2 (bvadd n (_ bv1 32))))))
9  (assert (forall ((n (_ BitVec 32)))
10          (=> (r2 n) (r1 (bvsub n (_ bv1 32))))))
11 (assert (forall ((n (_ BitVec 32)))
12          (=> (and (r2 n)
13                   (not (bvsge n (bvneg (_ bv10 32)))))
14              false)))
15
16 (check-sat)
```

Fig. 3: The example from Figure 1 in SMT-LIB notation, with bit-vector semantics.

arithmetic of width 32 is shown in Figure 3. Both sets of Horn clauses can easily be proven satisfiable by ELDARICA and other tools.

B. Related Work.

Horn solvers have been implemented using a variety of algorithms, often by extending methods from hardware or software model checking to the more general case of solving sets of Horn clauses. Existing state-of-the-art tools can be classified according to their underlying solving algorithm as the following:

- **CEGAR** and predicate abstraction, such as HSF [20], Duality [30], and ELDARICA;
- **IC3/PDR**, such as the PDR engine in Z3 [21]. The algorithm implemented in SPACER [28] extends IC3/PDR by maintaining both under- and over-approximations during analysis;
- **Transformation** of Horn clauses, such as VeriMAP [13] and Rahft [26];
- **Machine learning**, such as SynthHorn [33], FreqHorn [17] and HoIce [11], which progressively drive concrete invariant samples and use machine learning classification techniques to find the inductive invariant.

Many of the solvers in addition use techniques like abstract interpretation to synthesise invariants, and this way support the main algorithm.

Compared to other Horn solvers, distinguishing features of ELDARICA are the set of convergence heuristics implemented (Section II-C), which enable ELDARICA to solve particularly tricky Horn problems, the range of supported theories (including algebraic data types and bit-vectors), and the provided API.

II. An Overview of Eldarica

We start by describing the ELDARICA design and implementation. ELDARICA is open source, entirely implemented in Scala, and only depends on Java or Scala libraries,[2] which implies that ELDARICA can be used on any platform with a JVM. ELDARICA can be used as a standalone tool, but can also easily be integrated as a library into other systems implemented in Scala or Java. To reduce the JVM start-up/warm-up delay in standalone use, ELDARICA can also be run in a daemon mode.

ELDARICA uses PRINCESS [31] as SMT solver for satisfiability and implication checks, and as interpolation procedure for Presburger arithmetic [9], algebraic data-types [24], and bit-vectors [3]. The CEGAR engine of ELDARICA also loads PRINCESS as a library that provides the data-structures

[2]With the exception of the FLATA library optionally used for acceleration, as described below, which depends on Yices [16].

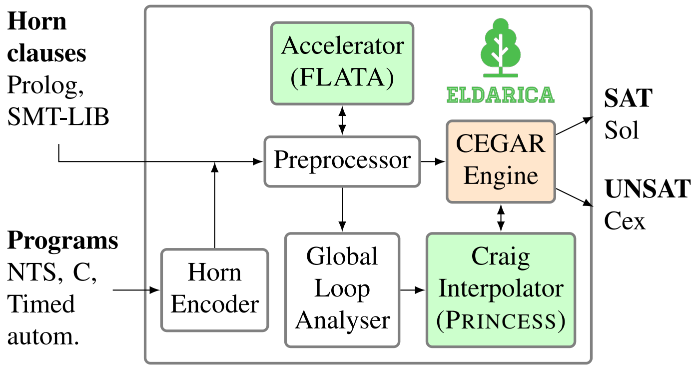


Fig. 4: The Main Architectural Components of ELDARICA

to represent terms, formulas, and background theories. This approach initially reduced the implementation effort, but also helps to speed up SMT queries because copying and conversion of expressions largely becomes unnecessary.

A. *Available Front-ends and Formats*

Figure 4 presents an overview of the ELDARICA's architecture. ELDARICA accepts input in a range of formats: the main input format for Horn clauses is (standard-compliant) SMT-LIB v2 [6], writing each clause as an explicitly quantified disjunction or implication. Support for the SMT-LIB rule dialect offered by Z3 is considered for the future. ELDARICA also supports Prolog-style input of Horn clauses over integer arithmetic.

ELDARICA is also able to parse programs in two (simple) formats, and handle the clause encoding internally. ELDARICA can read and verify input in the Numerical Transition Systems (NTS) format [1], [23], a format handled and produced by several verification tools. ELDARICA can also parse programs in a fragment of the C language (currently excluding pointers, arrays, and heap), as well as networks of timed automata in a C-like language with support for unbounded parallelism, clocks, binary communication channels, and time invariants [25].

B. *The Main Algorithms Used in* ELDARICA

To check the satisfiability of Horn clauses, ELDARICA applies lazy Cartesian predicate abstraction [19], [5], in combination with a variant of Counterexample-Guided Abstraction Refinement (CEGAR) [12], [4]. Horn clauses are first sent through a number of preprocessing stages, applying transformations such as (forward) slicing, (forward and backward) reachability analysis to eliminate dead relation symbols, clause inlining, splitting of clauses with complicated constraints or long bodies, constant propagation, abstract interpretation over an interval domain to infer basic information about variable ranges, as well as interval constraint propagation to further narrow down variable ranges.

The main CEGAR engine of ELDARICA then attempts to construct an abstract reachability (hyper)graph (ARG) that would witness satisfiability of Horn clauses, starting from a (user-provided, and often empty) set of predicates for each relation symbol. Implication properties are checked with the help of the SMT solver PRINCESS. If ARG construction fails, the obtained abstract counterexample DAG is checked for spuriousness by PRINCESS, resulting in either a concrete counterexample, or additional predicates computed through Craig interpolation. By default, ELDARICA maps the counterexample DAG to a tree interpolation problem for this purpose, but also disjunctive interpolation [32] can be switched on using the command-line option `-disj`.

C. *Convergence Heuristics*

Beyond basic CEGAR and Craig interpolation, ELDARICA applies two methods to minimise the likelihood of divergence, i.e., of the phenomenon that a model checker can sometimes fail to discover the right predicates, and continue refining the constructed abstraction indefinitely. The first method is based on *acceleration:* if during preprocessing cycles consisting of only linear clauses (with only conjunctive, numeric constraints) are detected, then precise static acceleration [10], [22] is applied to replace the cycle with a single clause with the same effect. ELDARICA loads the FLATA tool[3] as a Java library for this purpose. Acceleration later helps Craig interpolation to discover sufficiently general predicates, and has been shown to significantly extend the reach of CEGAR for tricky verification tasks [22]. Since this optimisation can sometimes slow down the model checker, and it is only applicable for cycles with linear clauses, it is optional and can be switched on with the command-line option `-stac`.

As a second method, ELDARICA uses *interpolation abstraction* [29] to control the predicates computed through Craig interpolation. Interpolation abstraction is driven by the results of a global analysis of the cycles (corresponding to program loops) present in a set of Horn clauses, including information about modified loop variables and strides of loop counters, derived during preprocessing. Among others, interpolation abstraction helps to analyse loops modifying multiple variables, e.g. the Horn clauses corresponding to the following program:

```
1 int x = 0, y = 0;
2 while (x < N) {
3   x++; y++;
4 }
5 assert(N < 0 || y == N);
```

In this case, loop analysis will identify the term `x - y` as a useful expression (or *interpolation template*) in invariants, and interpolation abstraction will guide the interpolation process towards expressions that avoid the variables `x`, `y`, unless they occur in the context `x - y`. This approach enables ELDARICA to rank interpolants according to their expected generality, and has been shown to speed up the solving process, as well as to significantly reduce the possibility of divergence [29], [14]. Interpolation templates can also be specified manually by the user to control the derived predicates.

Interpolation abstraction is enabled by default, but can optionally be switched off with the option `-abstract:off`.

[3] http://nts.imag.fr/index.php/Flata

There is also an option `-abstractPO` for running a portfolio of two solvers, one with interpolation abstraction enabled, and one without interpolation abstraction.

III. Status of Theory Support

A. Unbounded Integers

The development of Eldarica initially focused on the theory of unbounded linear integer arithmetic (LIA, quantifier-free Presburger arithmetic, but also including Booleans), for which efficient Craig interpolation is well understood. Among the supported theories, linear integer arithmetic in Eldarica is at this point the most refined and mature, and has been evaluated extensively in previous work [22], [29], [14].

Based on the interpolation procedure presented in [3], we have recently also added support for non-linear integer arithmetic (NIA) to Eldarica. The handling of NIA is best-effort though: procedures for NIA are necessarily incomplete, and quantifier-free interpolants do not exist in all cases. We have not yet collected a lot of experience with NIA problems.

B. Arrays

Eldarica can also handle problems with arrays, and can compute quantified solutions for such problems using the transformation approach from [8]. Eldarica accepts an extended Horn fragment for problems with arrays, with additional universal quantifiers allowed in front of each occurrence of a relation symbol specifying the intended quantifier structure of solutions. As an example, we consider a program filling an array with consecutive numbers:

```
1  int n, int ar[];
2  assume(n > 0);
3
4  int i = 0;
5  while (i < n) {
6    ar[i] = i;
7    i++;
8  }
9
10 assert(forall int j; 0 <= j && j < n => ar[j] >= 0);
```

A simple Horn representation of this verification task, using a single relation symbol inv representing the required loop invariant, is given in Figure 5. The encoding specifies that solutions are supposed to be of the form $inv(n, i, ar) = \forall ind.\ invM(n, i, ind, ar[ind])$, where the matrix $invM$ is the actual unknown to be determined by the Horn solver.

Instead of providing the quantifier pattern explicitly in the SMT-LIB input, it is also possible to leave the introduction of quantifiers to Eldarica, and simply declare inv to be a symbol with an array argument:

```
1  (declare-fun inv (Int Int (Array Int Int)) Bool)
```

The number of quantifiers to be introduced can be controlled using the command-line option `-arrayQuans:`n.

C. Algebraic Data-Types

Moving towards version 2, we have recently added support for algebraic data-types (ADTs) with fully-free constructors to Eldarica. This makes it possible to analyse Horn clauses

```
1  (set-logic HORN)
2
3  (declare-fun invM (Int Int Int Int) Bool)
4
5  (define-fun inv ((n Int) (i Int)
6                   (ar (Array Int Int))) Bool
7    (forall ((ind Int)) (invM n i ind (select ar ind))))
8
9  (assert (forall ((n Int) (ar (Array Int Int)))
10          (=> (> n 0) (inv n 0 ar))))
11
12 (assert (forall ((n Int) (i Int) (ar (Array Int Int)))
13          (=> (and (inv n i ar) (< i n))
14              (inv n (+ i 1) (store ar i i)))))
15
16 (assert (forall ((n Int) (i Int) (ar (Array Int Int)))
17          (=> (and (inv n i ar) (>= i n) )
18              (forall ((j Int))
19               (=> (and (<= 0 j) (< j n))
20                   (>= (select ar j) 0))))))
21
22 (check-sat)
```

Fig. 5: An array example in SMT-LIB. To solve the example using Eldarica, the option `-splitClauses` is needed.

```
1  (set-logic HORN)
2
3  (declare-datatype list ((nil)
4                          (cons (hd Int) (tl list))))
5
6  (declare-fun C (list list list) Bool)
7
8  (define-fun len ((l list)) Int (- (_size l) 1))
9
10 (assert (forall ((y list)) (C nil y y)))
11
12 (assert (forall ((x list) (y list) (r list) (i Int))
13          (=> (C x y r)
14              (C (cons i x) y (cons i r)))))
15
16 (assert (forall ((x list) (y list) (r list))
17          (=> (and (not (= r nil)) (C x y r))
18              (or (= (hd r) (hd x))
19                  (= (hd r) (hd y))))))
20
21 (assert (forall ((x list) (y list) (r list))
22            (=> (C x y r)
23                (= (len r) (+ (len x) (len y))))))
24
25 (check-sat)
```

Fig. 6: A list example in SMT-LIB.

with common data-types like enumerations, unions, tuples, lists, or trees. Clauses can also contain *size constraints*, i.e., reason about the number of occurrences of constructor symbols in a term.[4] This can be used to talk about the length of lists or the size of trees. ADTs are handled with the help of the decision and interpolation procedure presented in [24].

Figure 6 shows a Horn problem over the data-type of lists of integers. The data-type is defined with constructors `nil`, `cons`, and selectors `hd`, `tl`. The size of a list, in terms of the number of constructor symbols, can be accessed using the built-in operator `_size`; since `_size` also counts the `nil` operator, in line 8 we define a function `len` that computes

[4]SMT-LIB does currently not define a size operator for ADTs, so that resulting input is not SMT-LIB compliant.

standard list length. The relation symbol `C` is then defined to compute list concatenation, and in lines 16–23 two properties of concatenation are verified. A programmatic version of the example is provided in the next section.

At this point, ELDARICA is only able to compute quantifier-free (and recursion-free) solutions of Horn clauses over ADTs, which restricts the class of systems and properties that can meaningfully be analysed. For instance, ELDARICA cannot derive solutions that state sortedness of an unbounded list, or the property that all list elements are positive.

D. Bit-Vectors

ELDARICA version 2 also supports Horn clauses over bit-vectors, using a lazy encoding approach to map bit-vector constraints to quantifier-free Presburger constraints, which can then be solved and interpolated using the existing procedures in PRINCESS. The details of the interpolation procedure are described in a companion paper at FMCAD 2018 [3]. ELDARICA supports almost the full SMT-LIB bit-vector theory, although the interpolation procedure used for bit-vectors is optimised mainly for arithmetic constraints (as opposed to bit-wise operators) in Horn clauses. An SMT-LIB example with bit-vectors is given in Figure 3, and a programmatic example in the next section.

IV. PROGRAMMATIC USE OF ELDARICA

A. Algebraic Data Types

Since ELDARICA is implemented in Scala, it offers a convenient embedded domain-specific language for writing formulas and clauses, and can easily be integrated into other Scala applications. Integration into Java applications takes a similar form, but lacks the syntactic sugar provided through Scala, and at the moment requires the programmer to go through the slightly cumbersome process of calling Scala methods from Java. Formulas and data-types are constructed using the API of the underlying SMT solver PRINCESS.[5]

A complete runnable example is shown in Figure 7. In line 11, debugging assertions are switched off. In lines 13–17, again the ADT of lists over integers with sort name `list`, constructors `nil`, `cons`, and selectors `hd`, `tl` is defined (mutually recursive data-types can be created similarly). Lines 26–29 declare variables of sort integer and list, respectively, and line 31 a ternary relation symbol `C` over lists. The clauses in lines 34–35 are written in Prolog-like notation, and axiomatise `C` to represent concatenation. In line 39, a property about the head of a list resulting from concatenation is stated as a third clause. In line 41 the satisfiability of the three clauses is checked, with solution $C(x, y, r) \equiv y = r \vee hd(r) = hd(x)$.

To run the example, it is only necessary to have the Scala build tool `sbt` installed, which is included in many Linux distributions. Further dependencies, such as the Scala compiler and ELDARICA itself, will be downloaded automatically by the command `sbt run`.

[5]http://www.philipp.ruemmer.org/princess/doc/

```
1  // List-example.scala
2
3  import ap.SimpleAPI
4  import ap.theories.ADT
5  import lazabs.horn.bottomup._
6  import ADT._
7  import HornClauses._
8  import ap.parser.IExpression._
9
10 object ListExample extends App {
11   lazabs.GlobalParameters.get.assertions = false
12
13   val listADT = new ADT (Seq("list"),
14     Seq(("nil",  CtorSignature(Seq(), ADTSort(0))),
15         ("cons", CtorSignature(Seq(
16                         ("hd", OtherSort(Sort.Integer)),
17                         ("tl", ADTSort(0))),
18                   ADTSort(0)))))
19
20   val Seq(list)            = listADT.sorts
21   val Seq(nil, cons)       = listADT.constructors
22   val Seq(_, Seq(hd, tl))  = listADT.selectors
23
24   SimpleAPI.withProver { p =>
25     import p._
26
27     val n = createConstant("n", Sort.Integer)
28     val x = createConstant("x", list)
29     val y = createConstant("y", list)
30     val r = createConstant("r", list)
31
32     val C = createRelation("C", Seq(list, list, list))
33
34     val defClauses = List(
35       C(nil(), y, y)                  :- true,
36       C(cons(n, x), y, cons(n, r)) :- C(x, y, r)
37     )
38
39     val prop =
40       (hd(x) === hd(r) | hd(y) === hd(r)) :- (
41          C(x, y, r), r =/= nil())
42
43     SimpleWrapper.solve(prop :: defClauses) match {
44       case Left(sol) =>
45         println("sat"); println(sol mapValues (pp(_)))
46       case Right(cex) =>
47         println("unsat"); println(cex)
48     }
49   }
50 }
```

```
1 name := "list-example"
2 scalaVersion := "2.11.8"
3 resolvers += "uuverifiers" at "http://logicrunch.↩
     research.it.uu.se/maven/"
4 libraryDependencies += "uuverifiers" %% "eldarica" % "↩
     2.0-alpha2"
```

```
1 // Output of the program
2 > sbt run
3 [...]
4 sat
5 Map(C/3 -> _1 = _2 | hd(_2) = hd(_0))
6 [...]
```

Fig. 7: Runnable ELDARICA example, analysing Horn clauses over the data-type of lists. The program can be compiled and run with the command `sbt run`, which takes care of downloading all dependencies (including ELDARICA itself), compilation, and execution.

Benchmarks	#	Int		BV	
		ELDARICA sat/unsat	Z3 sat/unsat	ELDARICA sat/unsat	Z3 sat/unsat
Consistency	56	27/27	28/27	5/16	0/0
HOLA [15]	46	45/0	36/0	29/4	1/0
IntDualyzer	6	5/1	5/1	3/3	1/0
SLayer (chain.)	68	0/6	17/34	0/2	10/28
SLayer (fan.)	66	0/6	20/31	0/0	15/24
qarmc	13	9/1	11/1	5/1	0/0
ssh-simplified	23	13/8	9/9	13/6	1/0

Fig. 8: Results for ELDARICA 2.0-alpha3 and Z3 4.7.1 on integer and bit-vector benchmarks, an AMD Opteron 2220 SE machine, running 64-bit Linux and Java 1.8. Runtime was limited to 30min wall clock time, and heap space to 2GB. The table shows total number of benchmarks and the number of the benchmarks that each solver could solve.

B. Bit-vectors

We show an example of Horn clauses over bit-vectors in Figure 9. The overall structure of the program is similar as in the previous section. Bit-vector expressions are again constructed using the corresponding PRINCESS API, with the bit-vector operators provided in class `ModuloArithmetic`. The expression `bv(32, n)` generates the literal 32-bit constant n, while `bvadd` represents bit-vector addition. More generally, the bit-vector API offers access to the complete SMT-LIB bit-vector theory. The option `useTemplates` of the `SimpleWrapper` enables interpolation abstraction, which is in the API disabled by default.

V. EXPERIMENTAL RESULTS

Extensive experimental evaluations of ELDARICA have been published in multiple recent research papers [29], [14], we only report some experiments on some of the new features of ELDARICA version 2. Figure 8 shows a comparison of ELDARICA 2.0-alpha3[6] and Z3 4.7.1 on integer and bit-vector benchmarks. ELDARICA was run with the option `-abstractPO`, and Z3 with default options.

We use a collection of benchmarks in linear integer arithmetic from various sources.[7] C programs from HOLA [15] were first translated to NTS using Frama-C, and then to Horn clauses by ELDARICA. Since there are not many benchmarks for Horn clauses in bit-vector arithmetic, we wrote a script to convert all the operations in linear integer arithmetic to their equivalent bit-vector operations (32 bit signed). Using the script we transformed the original linear integer arithmetic benchmarks to bit-vector benchmarks. Of course, this can potentially change the satisfiability of the original benchmark, but it is useful for making a library of benchmarks of Horn clauses in bit-vector arithmetic.

The experiments show that ELDARICA performs well on most benchmark families. This might be due to the effective convergence heuristics in ELDARICA (Section II-C). An

[6]https://github.com/uuverifiers/eldarica/releases

[7]Benchmarks available at https://github.com/chc-comp/eldarica-misc and https://github.com/sosy-lab/sv-benchmarks/tree/master/clauses/

```
1  // BV-example.scala
2
3  import ap.SimpleAPI
4  import ap.theories.ModuloArithmetic._
5  import lazabs.horn.bottomup._
6  import HornClauses._
7  import ap.parser.IExpression._
8
9  object BVExample extends App {
10   lazabs.GlobalParameters.get.assertions = false
11
12   SimpleAPI.withProver { p =>
13     import p._
14
15     val x = createConstant("x", UnsignedBVSort(32))
16     val y = createConstant("y", UnsignedBVSort(32))
17
18     val C = createRelation("C",
19               Seq(UnsignedBVSort(32),
20                   UnsignedBVSort(32)))
21     val D = createRelation("D",
22               Seq(UnsignedBVSort(32),
23                   UnsignedBVSort(32)))
24
25     val defClauses = List(
26       C(bv(32, 1), bv(32, 1)) :- true,
27       C(bvadd(x, bv(32, 1)),
28         bvadd(bv(32, 1), y))  :- C(x, y),
29       D(x, y)                 :- (C(x, y),
30                                   x === bv(32, 0))
31     )
32
33     val prop =
34       (y === bv(32, 0)) :- D(x, y)
35
36     SimpleWrapper.solve(prop :: defClauses,
37                         useTemplates = true) match {
38       case Left(sol) =>
39         println("sat"); println(sol mapValues (pp(_)))
40       case Right(cex) =>
41         println("unsat"); println(cex)
42     }
43   }
44 }
```

```
1  name := "eldarica-example"
2  scalaVersion := "2.11.8"
3  resolvers += "uuverifiers" at "http://logicrunch.↩
     research.it.uu.se/maven/"
4  libraryDependencies += "uuverifiers" %% "eldarica" % "↩
     2.0-alpha2"
```

```
1  // Output of the program
2  > sbt run
3  [...]
4  sat
5  Map(C/2 -> _0 = _1, D/2 -> _1 = 0 & _0 = 0)
6  [...]
7  >
```

Fig. 9: Runnable ELDARICA example, analysing Horn clauses over bit-vectors. As in Figure 7, the program can be compiled and run with the command `sbt run`.

exception are the benchmarks in the SLayer families, which are solved more efficiently by Z3, possibly due to a large number of Boolean relation symbols arguments. Converting the problems to bit-vector semantics tends to produce harder benchmarks for both solvers. On many families ELDARICA can still solve a comparable number of problems, but generally fewer than with integer semantics.

VI. ADOPTION

ELDARICA has been used in a variety of applications, we list some examples. CoCoSim [2] is an analysis and code generation framework for Simulink that uses ELDARICA as one possible back-end. Similarly, JayHorn [27], a software model checking tool for Java supports ELDARICA as one of its back-ends. VAC [18] (Verifier of Access Control) an automatic tool for the analysis of Administrative Role Based Access Control (ARBAC) policies also relies on ELDARICA for solving Horn clauses. ELDARICA has also been used for the analysis of business processes expressed as Petri nets [29].

VII. CONCLUSIONS

ELDARICA is an efficient open source Horn solver supporting integer arithmetic, arrays, algebraic data types, and bit-vectors. It supports various input formats including SMT-LIB, Prolog, and numerical transition systems, and provides a Scala API. Future work includes (i) integration of further background theories, (ii) further improved heuristics to solve Horn clauses while avoiding divergence, (iii) generation of *quantified* solutions for problems with algebraic data types, and (iv) optimisation.

Acknowledgements: We thank Viktor Kuncak, Radu Iosif, Filip Konečný, and Pavle Subotic for their contributions to the ELDARICA project. We thank the reviewers for helpful comments. This work was supported by the Swedish Research Council (VR) under grant 2014-5484, and by the Swedish Foundation for Strategic Research (SSF) under the project WebSec (Ref. RIT17-0011).

REFERENCES

[1] http://nts.imag.fr.
[2] https://coco-team.github.io/cocosim/.
[3] Peter Backeman, Philipp Rümmer, and Aleksandar Zeljić. Bit-vector interpolation and quantifier elimination by lazy reduction. In *FMCAD*, 2018. To appear.
[4] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *ACM SIGPLAN PLDI*, 2001.
[5] Thomas Ball, Andreas Podelski, and Sriram K Rajamani. Boolean and Cartesian abstraction for model checking C programs. In *TACAS*, pages 268–283. Springer, 2001.
[6] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017.
[7] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, pages 24–51, 2015.
[8] Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. On solving universally quantified Horn clauses. In *SAS*, pages 105–125, 2013.
[9] Angelo Brillout, Daniel Kroening, Philipp Rümmer, and Thomas Wahl. An interpolating sequent calculus for quantifier-free Presburger arithmetic. *Journal of Automated Reasoning*, 47:341–367, 2011.
[10] N. Caniart, E. Fleury, J. Leroux, and M. Zeitoun. Accelerating interpolation-based model-checking. In *TACAS*, pages 428–442, 2008.
[11] Adrien Champion, Tomoya Chiba, Naoki Kobayashi, and Ryosuke Sato. ICE-based refinement type discovery for higher-order functional programs. In *TACAS*, pages 365–384, 2018.
[12] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
[13] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. VeriMAP: A tool for verifying programs through transformations. In *TACAS*, pages 568–574, 2014.
[14] Yulia Demyanova, Philipp Rümmer, and Florian Zuleger. Systematic predicate abstraction using variable roles. In *NFM*, pages 265–281, 2017.
[15] Isil Dillig, Thomas Dillig, Boyang Li, and Kenneth L. McMillan. Inductive invariant generation via abductive inference. In *OOPSLA*, pages 443–456, 2013.
[16] B. Dutertre and L. de Moura. The YICES SMT solver. http://yices.csl.sri.com/.
[17] Grigory Fedyukovich, Samuel J. Kaufman, and Rastislav Bodík. Sampling invariants from frequency distributions. In *FMCAD*, pages 100–107, Austin, TX, 2017.
[18] Anna Lisa Ferrara, P. Madhusudan, Truc L. Nguyen, and Gennaro Parlato. VAC — verifier of administrative role-based access control policies. In *CAV*, pages 184–191, 2014.
[19] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.
[20] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416. ACM, 2012.
[21] Krystof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *SAT*, pages 157–171, 2012.
[22] Hossein Hojjat, Radu Iosif, Filip Konecný, Viktor Kuncak, and Philipp Rümmer. Accelerating interpolants. In *ATVA*, pages 187–202, 2012.
[23] Hossein Hojjat, Filip Konecný, Florent Garnier, Radu Iosif, Viktor Kuncak, and Philipp Rümmer. A verification toolkit for numerical transition systems - tool paper. In *FM 2012*, pages 247–251, 2012.
[24] Hossein Hojjat and Philipp Rümmer. Deciding and interpolating algebraic data types by reduction. In *SYNASC*, 2017.
[25] Hossein Hojjat, Philipp Rümmer, Pavle Subotic, and Wang Yi. Horn clauses for communicating timed systems. In *HCVS*, pages 39–52, 2014.
[26] Bishoksan Kafle, John P. Gallagher, and José F. Morales. Rahft: A tool for verifying horn clauses using abstract interpretation and finite tree automata. In *CAV*, pages 261–268, 2016.
[27] Temesghen Kahsai, Philipp Rümmer, Huascar Sanchez, and Martin Schäf. JayHorn: A framework for verifying Java programs. In *CAV*, pages 352–358, 2016.
[28] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-based model checking for recursive programs. In *CAV*, pages 17–34, 2014.
[29] Jérôme Leroux, Philipp Rümmer, and Pavle Subotic. Guiding Craig interpolation with domain-specific abstractions. *Acta Inf.*, 53(4):387–424, 2016.
[30] Kenneth McMillan and Andrey Rybalchenko. Computing relational fixed points using interpolation. Technical report, January 2013. MSR-TR-2013-6.
[31] Philipp Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In *LPAR*, volume 5330 of *LNCS*, pages 274–289. Springer, 2008.
[32] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Disjunctive interpolants for Horn-clause verification. In *CAV*, volume 8044 of *LNCS*, pages 347–363. Springer, 2013.
[33] He Zhu, Stephen Magill, and Suresh Jagannathan. A data-driven CHC solver. In *ACM SIGPLAN PLDI*, pages 707–721, 2018.