A Verification Toolkit for Numerical Transition Systems

Tool Paper[*]

Hossein Hojjat[1], Florent Garnier[2], Radu Iosif[2],
Filip Konečný[2], Viktor Kuncak[1], and Philipp Rümmer[3]

[1] Swiss Federal Institute of Technology Lausanne (EPFL)
[2] Verimag, Grenoble, France
[3] Uppsala University, Sweden

Abstract. This paper reports on an effort to create benchmarks and a toolkit for rigorous verification problems, simplifying tool integration and eliminating ambiguities of complex programming language constructs. We focus on *Integer Numerical Transition Systems* (INTS), which can be viewed as control-flow graphs whose edges are annotated by Presburger arithmetic formulas. We describe the syntax, semantics, a front-end, and a first release of benchmarks for such transition systems. Furthermore, we present FLATA and ELDARICA, two new verification tools for INTS. The FLATA system is based on precise acceleration of the transition relation, while the ELDARICA system is based on predicate abstraction with interpolation-based counterexample-driven refinement. The ELDARICA verifier uses the PRINCESS theorem prover as a sound and complete interpolating prover for Presburger arithmetic. Both systems can solve several examples for which previous approaches failed and present a useful baseline for verifying integer programs. Our infrastructure is publicly available; we hope that it will spur further research, benchmarking, competitions, and synergistic communication between verification tools.

1 Introduction

Common representation formats, benchmarks, and tool competitions have helped research in a number of areas, including constraint solving, theorem proving, and compilers. To bring such benefits to the area of software verification, we are proposing a standardized logical format for programs, in terms of hierarchical infinite-state transition systems. The advantage of using a formally defined common format is avoiding ambiguities of programming language semantics and helping to separate semantic modeling from designing verification algorithms.

This paper focuses on systems whose transition relation is expressed in Presburger arithmetic. Integer Numerical Transition Systems, (denoted INTS throughout this paper), also known as counter automata, counter systems, or counter machines, are an infinite-state extension of the model of finite-state *boolean transition systems*, a model extensively used in the area of software verification [10]. The interest for INTS comes from the fact that they can encode various classes of systems with unbounded (or very large) data domains, such as hardware circuits, cache memories, or software systems with variables of non-primitive types, such as integer arrays, pointers and/or recursive data structures.

Any Turing-complete class of systems can, in principle be simulated by an INTS [12]. Despite this expressiveness, a number of recent works have revealed cost-effective approximate reductions of verification problems for several classes of complex systems to decision problems, phrased in terms on INTS. Examples of systems that can be effectively verified by means of integer programs include: specifications of hardware components [14], programs with singly-linked lists [2], trees [8], and integer arrays [3]. Hence the growing interest for analysis tools working on INTS.

[*] Supported in part by the Rich Model Toolkit initiative, `http://richmodels.org`

```
     var i: Int;
     var j: Int
l0:  havoc(i)
     assume(i >= 0)
l1:  havoc(j)
     assume(j >= 0)
l2:  var x: Int = i
     var y: Int = j
l3:  while (x != 0) {
l4:    x = x − 1
l5:    y = y − 1
     }
l6:  if (i == j)
l7:    assert (x == y)
```

(a)

$$\begin{tikzcd}
{} \arrow[r] & l_0 \arrow[d, "i' \geq 0"'] & & err \\
& l_1 \arrow[d, "j' \geq 0"'] & & l_7 \arrow[u, "x \neq y"'] \\
& l_2 \arrow[dr, "x' = i \wedge y' = j"'] & & l_6 \arrow[u, "i = j"'] \\
& & l_3 \arrow[ur, "x = 0"'] \arrow[dr, "x \neq 0"] & \\
& l_5 \arrow[ur, "y' = y - 1"] & & l_4 \arrow[ll, "x' = x - 1"]
\end{tikzcd}$$

(b)

Fig. 1. Example Program and its Numerical Transition System (NTS) Representation. By convention, if a variable v does not appear in the transition relation formula, we implicitly assume the frame condition $v = v'$ is conjoined with the formula.

We believe that robustly handling a wide variety of patterns of programming with integers is essential for building robust verification tools. As an example, consider the program in Figure 1(a). Most programmers would have little difficulty observing that the assertion will always succeed, but many tools, including non-relational abstract interpretation, as well as predicate abstraction with arbitrary interpolation can fail to prove the assertion to hold [11]. The integer numerical transition system for this program is in Figure 1(b). We have developed a toolkit for producing and manipulating such representations, as well as two very different analyzers that can analyze such transition systems. Both analyzers, ELDARICA and FLATA, in fact succeed for this, as well as for several other interesting examples. In other cases, however, our experiments show that these two techniques are complementary, so a user benefits from two different technologies that use the same input format.

2 The INTS Infrastructure

We have developed a toolkit for rigorous automated verification of programs in INTS format. The unifying component is the INTS library[4], which defines the syntax of the INTS representation by providing a parser and a library of abstract syntax tree classes. For the purposes of this paper, the INTS syntax can be considered to be just a textual description of a control flow graph labeled by Presburger arithmetic formulae, as e.g., the one in Figure 1 (b).
There are several components built using the INTS format, either as input, output or both. The INTS library is designed in order to support relatively easy bridging with new tools, which can be either front-ends (translators from mainstream programming languages into INTS), back-ends (verifier tools), or both. Currently, there exists tools to

[4] `http://richmodels.epfl.ch/ntscomp/ntslib`

generate INTS from sequential and concurrent C, Scala, and Verilog. We present two tools that can verify INTS programs: FLATA and Eldarica.

Flata verifier. FLATA[5] is a verification tool for hierarchical non-recursive INTS models. The verification technique used in FLATA is based on computing transitive closures of loops labeled by conjunctive transition relations. The three main classes of integer relations for which transitive closures can be computed precisely in finite time are: (1) *difference bounds constraints*, (2) *octagons*, and (3) *finite monoid affine transformations*. For these three classes, the transitive closures can be effectivelly defined in Presburger arithmetic.
The transitive closure computation is integrated in a semi-algorithmic method for computing the summary relation of an individual INTS. This algorithm builds the relation incrementally, by eliminating control states and composing incoming with outgoing relations. The verification method is *modular*: the tool computes the summary relation for each INTS independently of its calling context, thus avoiding the overhead of procedure inlining.

Eldarica verifier. ELDARICA implements predicate abstraction with Counter-Example Guided Abstraction Refinement (CEGAR). It generates an abstract reachability tree (ART) of the system on demand, using lazy abstraction [10] with Cartesian abstraction, and uses interpolation to refine the set of predicates [9]. ELDARICA uses caching of the previously explored states and formulae to prevent unnecessary reconstruction of tree. On demand, large block encoding [1] can be performed to reduce the number of calls to the interpolating theorem prover.
Eldarica refines abstractions with the help of *Craig Interpolants*, extracted from infeasibility proofs for spurious counterexamples. A complete interpolation procedure for (quantifier-free) Presburger arithmetic was proposed in [5], and has been implemented in the Princess prover [13] that we use in Eldarica. In this approach, interpolants are extracted from unsatisfiability proofs of conjunctive formulae, by recursively annotating the proof with partial and intermediate interpolants.

3 Experimental Comparison of the FLATA and ELDARICA Tools

We next show the benefits of INTS on comparing the performance of the FLATA and ELDARICA tools. This experimental analysis also points to the current strengths and weaknesses of the two tools we are developing. We have considered six sets of examples, extracted automatically from different sources: (a) C programs with arrays provided as examples of divergence in predicate abstraction [11], (b) INTS extracted from programs with singly-linked lists by the L2CA tool [2], (c) INTS extracted from VHDL models of circuits following the method of [14], (d) verification conditions for programs with arrays, expressed in the SIL logic of [3] and translated to INTS, (e) C programs provided as benchmarks in the NECLA static analysis suite, and (f) C programs with asynchronous procedure calls translated into INTS using the approach of [7] (the examples with extension .optim are obtained via an optimized translation method [6]).
The execution times are relative to an XXXX architecture. We noticed that the two tools behaved in a complementary way. In some cases (in particular on the (a) examples) the predicate abstraction method fails due to an unbounded number of loop unrollings required by refinement. In these cases, acceleration was capable to find the needed invariant rather quickly. On the other hand, in particular on the (f) examples, the acceleration

[5] `http://www-verimag.imag.fr/FLATA.html`

approach was unsuccessful in reducing loops with linear but non-octagonal relations. In these cases, the predicate abstraction found the needed Presburger invariants for proving correctness, and error traces, for the erroneous examples.

Model	Time [s]	
	Flata	Eld.
(a) Examples from [11]		
anubhav (C)	0.4	2.5
copy1 (E)	1.0	9.0
cousot (C)	0.3	-
loop1 (C)	0.3	2.4
loop (C)	0.3	1.3
scan (E)	1.5	-
string_concat1 (E)	3.1	-
string_concat (E)	3.0	-
string_copy (E)	2.6	-
substring1 (E)	0.3	0.7
substring (E)	1.3	0.7
(b) Examples from L2CA [2]		
bubblesort (E)	9.6	0.6
insdel (E)	0.1	0.1
insertsort (E)	1.4	0.6
listcounter (C)	0.2	-
listcounter (E)	0.2	0.3
listreversal (C)	3.1	0.4
listreversal (E)	0.4	0.6
mergesort (E)	0.8	0.6
selectionsort (E)	0.9	0.3

Model	Time [s]	
	Flata	Eld.
(c) VHDL models from [14]		
counter (C)	0.1	2.2
register (C)	0.1	1.3
synlifo (C)	14.7	57.3
(d) Verification conditions for array programs [3]		
rotation_vc.1 (C)	0.5	2.3
rotation_vc.2 (C)	0.8	2.4
rotation_vc.3 (C)	0.7	0.4
rotation_vc.1 (E)	0.6	1.5
split_vc.1 (C)	3.1	3.2
split_vc.2 (C)	2.2	2.4
split_vc.3 (C)	2.1	0.6
split_vc.1 (E)	19.0	2.3
(e) NECLA benchmarks		
blast (C)	0.2	3.3
inf1 (E)	0.1	1.4
inf4 (E)	0.6	4.8
inf6 (C)	0.0	3.0
inf8 (C)	0.2	4.7

Model	Time [s]	
	Flata	Eld.
(f) Examples from [7]		
h1 (E)	-	13.8
h1.optim (E)	0.5	2.7
h1h2 (E)	-	35.3
h1h2.optim (E)	0.7	5.2
simple (E)	-	11.9
x simple.optim (E)	0.5	2.7
test0 (C)	-	36.2
test0.optim (C)	0.2	6.0
test0 (E)	-	13.0
test0.optim (E)	0.4	2.7
test1.optim (C)	0.4	10.7
test1.optim (E)	0.9	8.3
test2_1.optim (E)	0.9	5.9
test2_2.optim (E)	1.8	7.7
test2.optim (C)	5.2	46.0
wrpc.manual (C)	2.3	1.8
wrpc (E)	-	14.7
wrpc.optim (E)	-	3.7

Fig. 2. Benchmarks for **Flata** and **Eld**arica. The letter after the model name distinguishes **C**orrect from **E**rratic models. Items with "-" led to a timeout for the respective tool.

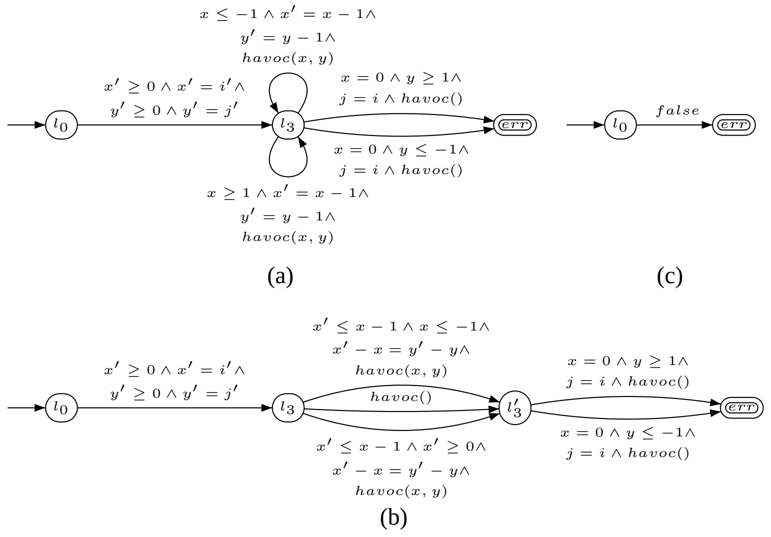
References

1. D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *FMCAD*, 2009.
2. A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. In *CAV*, pages 517–531, 2006.
3. M. Bozga, P. Habermehl, R. Iosif, F. Konečný, and T. Vojnar. Automatic verification of integer array programs. In *CAV*, pages 157–172, 2009.
4. M. Bozga, R. Iosif, and F. Konečný. Fast acceleration of ultimately periodic relations. In *CAV*, pages 227–242, 2010.
5. A. Brillout, D. Kroening, P. Rümmer, and T. Wahl. An interpolating sequent calculus for quantifier-free Presburger arithmetic. In *IJCAR*, LNCS. Springer, 2010.
6. P. Ganty. Personal communication.
7. P. Ganty and R. Majumdar. Algorithmic verification of asynchronous programs. *CoRR*, abs/1011.0551, 2010.
8. P. Habermehl, R. Iosif, A. Rogalewicz, and T. Vojnar. Proving termination of tree manipulating programs. In *ATVA*, pages 145–161, 2007.
9. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244. ACM, 2004.

10. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.
11. R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *TACAS*, pages 459–473, 2006.
12. M. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, 1967.
13. P. Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In *LPAR*, volume 5330 of *LNCS*, pages 274–289. Springer, 2008.
14. A. Smrcka and T. Vojnar. Verifying parametrised hardware designs via counter automata. In *Haifa Verification Conference*, pages 51–68, 2007.

A Demonstration of Flata on the Running Example

Flata verifies the program in Figure 1(a) by computing its transfer function. The method is similar to the classical conversion of finite automata into regular expressions. We start by eliminating control states in Figure 1(b) which have no self-loop. Each such elimination requires to compose all incoming with all outgoing edge relations. Figure 3(a) shows the result after elimination of control states l_1, l_2, l_4, l_5, l_6, and l_7.
Next, we eliminate control states with self-loops. Each such elimination requires the computation of the transitive and reflexive closure of a (possibly disjunctive) relation (one disjunct for each self-loop relation). For this, we apply a semi-algorithm which is based on the algorithm from [4] computing transitive and reflexive closure of difference bounds and octagonal relations. When successful, we replace the loop with meta-transitions labeled with the computed closure. Elimination of l_3 in Figure 3(a) yields meta-transitions shown in Figure 3(b). Finally, we eliminate l_3 and l_3', now without self-loops, which gives an inconsistent transfer function in Figure 3(c), thus proving correctness of the program.



$x \leq -1 \wedge x' = x - 1 \wedge$
$y' = y - 1 \wedge$
$havoc(x, y)$
$x' \geq 0 \wedge x' = i' \wedge$
$y' \geq 0 \wedge y' = j'$
$x = 0 \wedge y \geq 1 \wedge$
$j = i \wedge havoc()$
l_0
l_3
err
$x = 0 \wedge y \leq -1 \wedge$
$j = i \wedge havoc()$
$x \geq 1 \wedge x' = x - 1 \wedge$
$y' = y - 1 \wedge$
$havoc(x, y)$
l_0
$false$
err
(a)
(c)
$x' \leq x - 1 \wedge x \leq -1 \wedge$
$x' - x = y' - y \wedge$
$havoc(x, y)$
$havoc()$
$x' \geq 0 \wedge x' = i' \wedge$
$y' \geq 0 \wedge y' = j'$
l_0
l_3
l_3'
err
$x = 0 \wedge y \geq 1 \wedge$
$j = i \wedge havoc()$
$x = 0 \wedge y \leq -1 \wedge$
$j = i \wedge havoc()$
$x' \leq x - 1 \wedge x' \geq 0 \wedge$
$x' - x = y' - y \wedge$
$havoc(x, y)$
(b)

Fig. 3. Deciding Safety by Elimination of Control Locations

B Demonstration of Eldarica on the Running Example

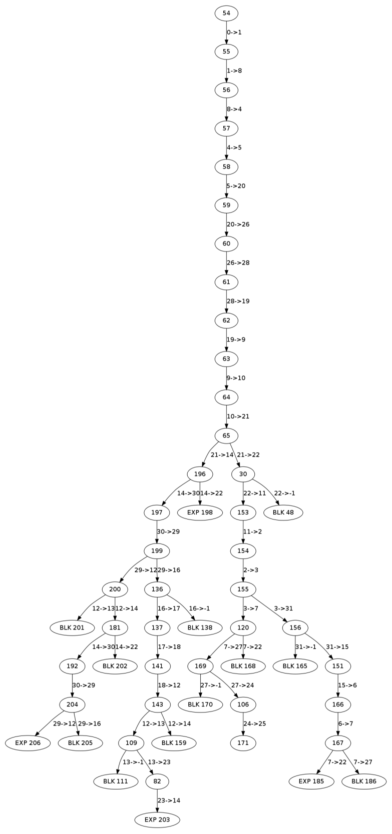


Fig. 4. Abstract reachability tree for blast.nts

C Example for Interpolation

As an example for interpolation, consider the infeasible path $l_0, l_1, l_2, l_3, l_6, l_7, err$ from our example program in Figure 1. By converting the statements and guards into a formula, Eldarica extracts the following path constraint:

$$\underbrace{i_0 = 0 \land j_0 = 0 \land i_1 \geq 0 \land j_1 \geq 0 \land x_0 = i_1 \land y_0 = j_1 \land x_0 = 0}_{\phi(i_0,j_0,i_1,j_1,x_0,y_0)} \land \underbrace{i_1 = j_1 \land x_0 \neq y_0}_{\psi(i_1,j_1,x_0,y_0)}$$

Princess derives the inconsistency of constraints like this by linear combination of the equations, as shown in Figure 5, forming the unsatisfiable consequence $0 \neq 0$. For the given partitioning of the constraint into $\phi(i_0, j_0, i_1, j_1, x_0, y_0), \psi(i_1, j_1, x_0, y_0)$, an interpolant can be computed by projecting this linear combination to the equations originating from the left partition:

$$I(i_1, j_1, x_0, y_0) \equiv -1 \cdot (x_0 - i_1 = 0) + 1 \cdot (y_0 - j_1 = 0) \equiv (y_0 - x_0 + i_1 - j_1 = 0)$$

The resulting predicate, $i_1 - j_1 = x_0 - y_0$, enables Eldarica to refine the abstract reachability tree and construct an inductive invariant for the loop in the example program, proving its safety.

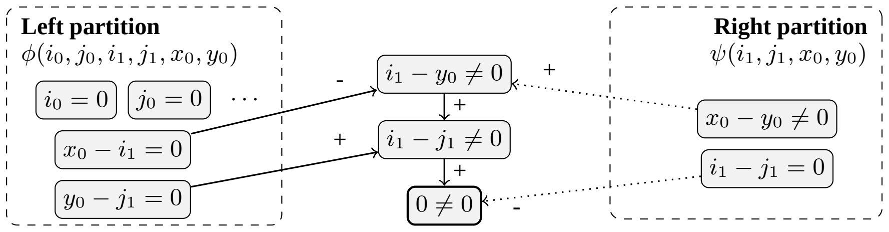


Fig. 5. Proof about path constraints. All atoms are normalised to have right-hand side 0.