A Verification Toolkit for Numerical Transition Systems

Tool Paper*

Hossein Hojjat[1], Filip Konečný[2,4], Florent Garnier[2],
Radu Iosif[2], Viktor Kuncak[1], and Philipp Rümmer[3]

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

Abstract. This paper presents a publicly available toolkit and a benchmark suite for rigorous verification of *Integer Numerical Transition Systems* (INTS), which can be viewed as control-flow graphs whose edges are annotated by Presburger arithmetic formulas. We present FLATA and ELDARICA, two 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. The infrastructure is a starting point for rigorous benchmarking, competitions, and standardized communication between tools.

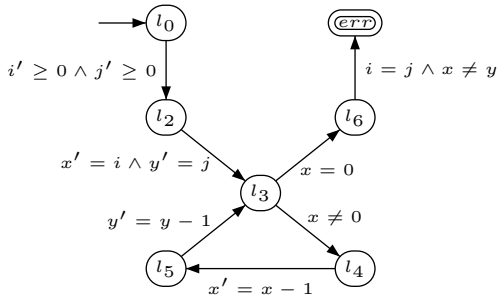
1 Introduction

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 in 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 [8]. 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

* Supported by the Rich Model Toolkit initiative, `http://richmodels.org`, the Czech Science Foundation (projects P103/10/0306 and 102/09/H042), the Czech Ministry of Education (COST OC10009 and MSM 0021630528), the EU/CzechIT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070, the BUT project FIT-12-1 and the Microsoft Innovation Cluster for Embedded Software.

```
    var i,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) assert(x == y)
```

(a)



(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 that the frame condition $v = v'$ is conjoined. The states l_1 and l_2 have been merged in the NTS.

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. 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 [10], programs with singly-linked lists [1], trees [6], and integer arrays [2].

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 [9]. 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 example, as well as for several other interesting examples. Our experiments show that the two tools are complementary in general, so users benefit from different techniques 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[5], 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 is considered to be a textual description of a control flow graph labeled by Presburger arithmetic formulae, as in Figure 1 (b).

At this point, there are several tools supporting the INTS format, as input and/or output language. The INTS library is designed for easy bridging with new tools, which can be either front-ends (translators from mainstream programming languages into INTS),

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

back-ends (verification tools), or both. Currently, there exist tools to 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[6] is a verification tool for hierarchical non-recursive INTS models. The tool computes the summary relation for each INTS independently of its calling context, thus avoiding the overhead of procedure inlining. The verification is based on computing transitive closure of loops. Classes of integer relations for which transitive closures can be computed precisely include: (1) *difference bounds relations*, (2) *octagons*, and (3) *finite monoid affine transformations*. For these three classes, the transitive closures can be effectively defined in Presburger arithmetic. FLATA integrates the transitive closure computation method for difference bounds and octagonal relations from [3] in a semi-algorithm computing the summary relation incrementally, by eliminating control states and composing incoming with outgoing relations.

Eldarica verifier. ELDARICA[7] 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 with Cartesian abstraction, and uses interpolation to refine the set of predicates [7]. For checking the feasibility of paths, and constructing abstractions, ELDARICA employs the provers Z3[8] and Princess.[9] In addition, ELDARICA uses caching of previously explored states and formulae to prevent unnecessary reconstruction of trees. Large block encoding 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. The complete interpolation procedure for Presburger arithmetic was proposed in [4], and is implemented as part of Princess.

3 Experimental Comparison of the FLATA and ELDARICA Tools

We next give an experimentally compare FLATA and ELDARICA on six sets of examples extracted automatically from different sources: (a) C programs with arrays provided as examples of divergence in predicate abstraction [9], (b) INTS extracted from programs with singly-linked lists by the L2CA tool [1], (c) INTS extracted from VHDL models of circuits following the method of [10], (d) verification conditions for programs with arrays, expressed in the SIL logic of [2] 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 [5] (the examples with extension .optim are obtained via an optimized translation method). Experiments were ran on an Intel®Core™2 Duo @ 2.66GHz with 3GB RAM. The two tools behaved in a complementary way. In some cases (examples (a)) 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 (examples (f)), the acceleration approach was unsuccessful in reducing

[6] `http://www-verimag.imag.fr/FLATA.html`
[7] `http://lara.epfl.ch/w/eldarica`
[8] `http://research.microsoft.com/en-us/um/redmond/projects/z3/`
[9] `http://www.philipp.ruemmer.org/princess.shtml`

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 [9]		
anubhav (C)	0.8	2.0
copy1 (E)	1.8	13.9
cousot (C)	12.0	-
loop1 (E)	1.3	12.0
loop (E)	1.9	10.6
scan (E)	2.5	-
string_concat1 (E)	4.7	-
string_concat (E)	4.7	-
string_copy (C)	0.4	-
substring1 (E)	0.6	5.5
substring (E)	1.6	0.7
(b) Examples from L2CA [1]		
bubblesort (E)	14.1	2.5
insdel (E)	0.1	0.3
insertsort (E)	1.9	0.8
listcounter (C)	0.3	-
listcounter (E)	0.3	0.3
listreversal (C)	4.8	0.6

Model	Time [s] Flata	Eld.
(c) VHDL models from [10]		
counter (C)	0.1	1.7
register (C)	0.2	1.2
synlifo (C)	16.4	20.3
(d) Verification conditions for array programs [2]		
rotation_vc.1 (C)	0.8	2.0
rotation_vc.2 (C)	1.1	2.2
rotation_vc.3 (C)	1.2	0.3
rotation_vc.1 (E)	1.1	1.4
split_vc.1 (C)	3.8	3.0
split_vc.2 (C)	2.8	2.2
split_vc.3 (C)	2.6	0.6
split_vc.1 (E)	30.2	2.2
(e) NECLA benchmarks		
inf1 (E)	0.2	0.4
inf4 (E)	0.9	0.6
inf6 (C)	0.1	0.4
inf8 (C)	0.3	0.6

Model	Time [s] Flata	Eld.
(f) Examples from [5]		
h1 (E)	-	5.7
h1.optim (E)	0.6	1.3
h1h2 (E)	-	19.0
h1h2.optim (E)	0.9	4.3
simple (E)	-	6.1
simple.optim (E)	0.6	1.3
test0 (C)	-	30.6
test0.optim (C)	0.3	5.3
test0 (E)	-	5.0
test0.optim (E)	0.6	1.3
test1.optim (C)	0.6	8.5
test1.optim (E)	1.4	6.8
test2_1.optim (E)	1.2	4.6
test2_2.optim (E)	2.8	4.6
test2.optim (C)	6.3	72.9
wrpc.manual (C)	0.6	1.2
wrpc (E)	-	9.5
wrpc.optim (E)	-	3.0

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

References

1. 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.
2. M. Bozga, P. Habermehl, R. Iosif, F. Konečný, and T. Vojnar. Automatic verification of integer array programs. In *CAV*, pages 157–172, 2009.
3. M. Bozga, R. Iosif, and F. Konečný. Fast acceleration of ultimately periodic relations. In *CAV*, pages 227–242, 2010.
4. A. Brillout, D. Kroening, P. Rümmer, and T. Wahl. An interpolating sequent calculus for quantifier-free Presburger arithmetic. In *IJCAR*, LNCS. Springer, 2010.
5. P. Ganty and R. Majumdar. Algorithmic verification of asynchronous programs. *CoRR*, abs/1011.0551, 2010.
6. P. Habermehl, R. Iosif, A. Rogalewicz, and T. Vojnar. Proving termination of tree manipulating programs. In *ATVA*, pages 145–161, 2007.
7. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244. ACM, 2004.
8. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.
9. R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *TACAS*, pages 459–473, 2006.
10. A. Smrcka and T. Vojnar. Verifying parametrised hardware designs via counter automata. In *Haifa Verification Conference*, pages 51–68, 2007.