

The Eureka Tool for Software Model Checking*

Alessandro Armando
AI-Lab, DIST
Università di Genova, Italy

Massimo Benerecetti
Dip. di Scienze Fisiche
Università “Federico II”
Napoli, Italy

Dario Carotenuto
Dip. di Scienze Fisiche
Università “Federico II”
Napoli, Italy

Jacopo Mantovani
AI-Lab, DIST
Università di Genova, Italy

Pasquale Spica
Dip. di Scienze Fisiche
Università “Federico II”
Napoli, Italy

ABSTRACT

We describe EUREKA, a symbolic model checker for Linear Programs with arrays, i.e. programs where variables and array elements range over a numeric domain and expressions involve linear combinations of variables and array elements. This language fragment easily encodes a large class of programs for which, as demonstrated by our experiments, techniques based on predicate abstraction do not apply successfully.

1. INTRODUCTION

EUREKA is a software model checker for Linear Programs with arrays [1], a fragment of the C programming language where variables and array elements range over a numeric domain and expressions involve linear combinations of variables and array elements. EUREKA is targeted towards the verification of reachability properties which can be specified by adding assertions to the program. A number of features are supported, among which arbitrarily nested loops and non-determinism.

EUREKA interprets the counterexample guided abstraction refinement (CEGAR for short) paradigm in a novel way by using array indexes instead of predicates. In the most common approaches (e.g. SLAM [2], BLAST [6], SATABS [5]) a program is abstracted w.r.t. a set of predicates, the abstraction is a Boolean Program, and refinement searches for new predicates in order to build a new, more refined abstraction. Unlike these approaches, EUREKA abstracts the program w.r.t. a family of sets of array indexes, the abstraction is a Linear Program (without arrays), and refinement searches for new array indexes.

*The work described in this paper has been partially supported by the Italian Ministry of University and Research within the the PRIN project no. 2003097383_002, “Synthesis of deduction-based decision procedures with applications to the automatic formal analysis of software”.

The ability to analyse Linear Programs with arrays is particularly important as arithmetic and arrays are ubiquitous in programming and many real-world programs belong to this class. Moreover, most predicate abstraction techniques (e.g. SLAM and BLAST) suffer from a severe lack of precision when dealing with arrays. The experimental analysis of Section 3 demonstrates the effectiveness and scalability of the EUREKA approach when compared with other state-of-the-art tools based on predicate abstraction.

2. LINEAR ABSTRACTION REFINEMENT

Let P be a Linear Program with arrays, and let R be an array-indexed family of sets of array indexes. The CEGAR procedure implemented in EUREKA amounts to iterating the following steps.

Abstraction. Let a be an array and $[k_1, \dots, k_n]$ be a permutation of the elements in $R(a) \in R$. If l is a linear expression, then an abstraction of l w.r.t. R , say \hat{l} , is obtained from l by replacing every expression of the form $a[e]$ with $\text{abs}(a[e], [k_1, \dots, k_n])$, where

$$\begin{aligned} \text{abs}(a[e], []) &= u \\ \text{abs}(a[e], [k_1, k_2, \dots, k_n]) &= (\hat{e} == k_1 ? a^{k_1} : \text{abs}(a[e], [k_2, \dots, k_n])), \end{aligned}$$

each a^{k_i} is a numeric variable denoting the value of the k_i -th element of a (for $i = 1, \dots, n$), and u is a constant denoting an undefined value. An abstraction \hat{P} w.r.t. R is obtained from P by replacing all the expressions l occurring in P with \hat{l} , and then by replacing every assignment of the form $a[e_1] = e_2$; with the (parallel) assignment

$$a^{k_1}, \dots, a^{k_n} = (\hat{e}_1 == k_1 ? \hat{e}_2 : a^{k_1}), \dots, (\hat{e}_1 == k_n ? \hat{e}_2 : a^{k_n});.$$

If $R(a) = \emptyset$, then the parallel assignment reduces to a skip (;) statement.

Model Checking. The resulting Linear Program (without arrays) \hat{P} is then model-checked by using the interprocedural data-flow analysis described in [1]. If \hat{P} is found to be safe, the computation stops reporting that P is safe. Otherwise, an abstract error trace τ is computed.

Simulation. The error trace τ of \hat{P} found in the previous step is symbolically executed in P to check its feasibility. This is done by building a set of quantifier-free formulæ $\Phi(\tau)$ whose satisfiability (w.r.t. the union of the theory of arrays and Linear Arithmetic) guarantees the executability of τ in P . If $\Phi(\tau)$ is found to be satisfiable, then τ is reported to be

an error trace of P and the procedure halts, otherwise the proof of unsatisfiability Π of $\Phi(\tau)$ is fed to the next phase.

Refinement. The proof of unsatisfiability Π is inspected and R extended in such a way to rule out τ from the execution traces of the refined program.

3. COMPARATIVE EXPERIMENTS

We have tested EUREKA against a number of problems that involve reasoning on both arithmetic and arrays and thus allow to thoroughly assess the effectiveness and scalability properties of our tool. On the same problems we have tested two well-known symbolic model checkers that employ predicate abstraction, namely BLAST and SATABS. The experiments have been carried out on a 2.4GHz Pentium IV running Linux with memory limit set to 800MB and time limit set to 30 minutes. An excerpt of the results of our experiments is given in Tables 1, 2, and 3

The first two benchmark problems involve string manipulation, the GRAY CODE problem consists in an implementation of the (n, k) -Gray code algorithm [3]. As revealed by the results of our experiments, sorting algorithms like the BUBBLE SORT, SELECTION SORT, and INSERTION SORT constitute a hard testbench for the tools due to the tight coupling of data and control. The FIBONACCI problem iteratively computes and sums the first N Fibonacci numbers. This problem is a slight variation of the one comprised in the SNU Real-Time benchmark suite [7]. Also, we tested the tools against an implementation of the BRESENHAM problem, a well-known algorithm initially developed with the purpose of drawing lines with digital plotters [4] and later used for computer displays. SWAP is a program that iteratively calls a procedure that swaps the values of two variables. The latter benchmark can be found in the BLAST source code distribution. FIBONACCI, BRESENHAM, and SWAP are the only benchmark problems that do not involve reasoning on arrays.

All problems are parametric in a positive integer N . The size of the arrays occurring in the programs and/or the number of iterations carried out by the loops increase as N increases. Thus the higher is the value of N , the bigger is the search space to be analysed. Each entry of the tables shows the greatest instance the tools are able to analyse and the time in seconds. Also, we give the time taken by the refinement phase of SATABS, the number of array elements found by EUREKA during the refinement phase, and the sum of the sizes of the arrays involved in the programs. Numbers with * indicate that the tool can analyse greater instances than the one shown. All benchmark families but STRING COMPARE are safe.

As shown by Table 1, on most problems BLAST reports an incorrect answer, that is, it concludes that the program is unsafe when it is safe instead. The reason of this lies in that different array elements are indistinguishable for BLAST. By default, SATABS allows 50 iterations of the abstract-check-refine loop. We increased this number to 100 with option `-iterations`. The inconclusive outcome in Table 2 means that SATABS is not able to output a result after 100 iterations. The results of the experiments demonstrate the great effort required by the refinement phase despite the efficiency of current SAT solvers. The experiments with EUREKA (see Table 3) confirm the effectiveness of the Linear Abstraction and reveal the difficulties of the approaches

Table 1: BLAST experimental results.

| Benchmark | BLAST | |
|----------------|-------|------------|
| | N | Total Time |
| STRING COPY | | Incorrect |
| STRING COMPARE | 100* | 435.26 |
| GRAY CODE | | Incorrect |
| PARTITION | | Incorrect |
| BUBBLE SORT | | Incorrect |
| INSERTION SORT | | Incorrect |
| SELECTION SORT | | Incorrect |
| FIBONACCI | 24 | (5.68) |
| BRESENHAM | | Error |
| SWAP | 300* | 782.25 |

Table 2: SATABS experimental results.

| Benchmark | SATABS | | |
|----------------|--------|-----------------|------------|
| | N | Refinement Time | Total Time |
| STRING COPY | 10 | 105.98 | 144.69 |
| STRING COMPARE | 12 | 292.19 | 348.19 |
| GRAY CODE | | Inconclusive | |
| PARTITION | | Inconclusive | |
| BUBBLE SORT | 2 | 24.39 | 30.42 |
| INSERTION SORT | 2 | 51.43 | 74.74 |
| SELECTION SORT | 2 | 75.53 | 115.86 |
| FIBONACCI | 1000* | 1.65 | 2.88 |
| BRESENHAM | 1000* | 71.15 | 83.16 |
| SWAP | 64 | 8.25 | 109.44 |

based on predicate abstraction when dealing with programs featuring a tight interplay between arithmetic and array manipulation. Particularly, EUREKA performs best when few array elements are needed to prove the property given.

Table 3: EUREKA experimental results.

| Benchmark | EUREKA | | |
|----------------|--------|------------|---|
| | N | Total Time | $\frac{re_{jned}}{total}$ array elements |
| STRING COPY | 1000* | 134.63 | 1/2N |
| STRING COMPARE | 1000* | 18.11 | 1/2N |
| GRAY CODE | 60 | 101.31 | 16/28 |
| PARTITION | 40 | 111.16 | 1/N |
| BUBBLE SORT | 9 | 92.29 | N/N |
| INSERTION SORT | 16 | 64.55 | N/N |
| SELECTION SORT | 9 | 58.41 | N/N |
| FIBONACCI | 1000* | 7.45 | 0/0 |
| BRESENHAM | 1000* | 11.25 | 0/0 |
| SWAP | 1000* | 2.45 | 0/0 |

4. REFERENCES

- [1] A. Armando, M. Benerecetti, and J. Mantovani. Model checking linear programs with arrays. In *SoftMC*, volume 144 of *ENTCS*. Elsevier, 2005.
- [2] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN*, volume 2057 of *LNCS*. Springer, 2001.
- [3] P. E. Black. Gray code, in Dictionary of Algorithms and Data Structures. See www.nist.gov/dads/HTML/graycode.html, 2005.
- [4] J. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.
- [5] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *TACAS*, volume 3440 of *LNCS*. Springer, 2005.
- [6] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*. ACM Press, 2002.
- [7] Seoul National University, Real Time Research Group. SNU Real Time Benchmarks. See archi.snu.ac.kr/realtime/benchmark.