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 [3], BLAST [4], 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 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 $[k_1, \ldots, k_n]$ 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 $\widehat{l}$, is obtained from l by replacing every expression of the form $a\texttt{[}e\texttt{]}$ with $\text{abs}(a\texttt{[}e\texttt{]}, [k_1, \ldots, k_n])$, where

$$\begin{array}{l}\text{abs}(a\texttt{[}e\texttt{]}, []) = \mathfrak{u}\\ \text{abs}(a\texttt{[}e\texttt{]}, [k_1, k_2, \ldots, k_n]) = (\widehat{e} \texttt{==} k_1 \texttt{ ? } a^{k_1} \texttt{ : } \text{abs}(a\texttt{[}e\texttt{]}, [k_2, \ldots, k_n])),\end{array}$$

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

$$a^{k_1}, \ldots, a^{k_n} \texttt{=} (\widehat{e}_1 \texttt{==} k_1 \texttt{ ? } \widehat{e}_2 \texttt{ : } a^{k_1}), \ldots, (\widehat{e}_1 \texttt{==} k_n \texttt{ ? } \widehat{e}_2 \texttt{ : } a^{k_n})\texttt{;}.$$

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

Model Checking. The resulting Linear Program (without arrays) $\widehat{P}$ is then model-checked by using the interprocedural data-flow analysis described in [1]. If $\widehat{P}$ 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 $\widehat{P}$ 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

[*]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”.

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 [11]. 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 [12]. 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 [13] 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 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 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

Table 3: EUREKA experimental results.

Benchmark	EUREKA		
	N	Total Time	refined/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'05*, ser. ENTCS, vol. 144. Elsevier, 2005.

[2] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *CAV*, 2000.

[3] T. Ball and S. K. Rajamani, "Automatically validating temporal safety properties of interfaces," in *SPIN*. Springer, 2001.

[4] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *POPL*. ACM Press, 2002.

[5] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "SATABS: SAT-based predicate abstraction for ANSI-C," in *TACAS*, ser. LNCS, vol. 3440. Springer, 2005.

[6] T. Reps, S. Horwitz, and M. Sagiv, "Precise

interprocedural dataflow analysis via graph reachability," in *POPL*. ACM Press, 1995.

[7] A. Armando, M. Benerecetti, and J. Mantovani, "Abstraction refinement of linear programs with arrays," in *TACAS*, ser. LNCS, vol. 4424. Springer, 2007.

[8] R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill, "Possibly not closed convex polyhedra and the parma polyhedra library," in *SAS*, ser. LNCS, vol. 2477. Springer, 2002.

[9] C. Barrett and S. Berezin, "CVC Lite: A new implementation of the cooperating validity checker," in *CAV*, ser. LNCS. Springer, 2004.

[10] P. J. Plauger, "The standard template library," *C/C++ Users Journal*, vol. 13, no. 12, 1995.

[11] P. E. Black, "Gray code, in dictionary of algorithms and data structures," 2005, see http://www.nist.gov/dads/HTML/graycode.html.

[12] Seoul National University, Real Time Research Group, "SNU Real Time Benchmarks," available at http://archi.snu.ac.kr/realtime/benchmark.

[13] J. Bresenham, "Algorithm for computer control of a digital plotter," *IBM Systems Journal*, vol. 4, no. 1, pp. 25–30, 1965.

[14] A. Armando, C. Castellini, and J. Mantovani, "Software model checking using linear constraints," in *ICFEM'04*, ser. LNCS, vol. 3308. Springer, 2004.

[15] *Minimum Operational Performance Standards for Traffic Alert and Collision Avoidance System II (TCAS II) Airborne Equipment*, Radio Technical Commission for Aeronautics (RTCA), Inc., 1997, document no. DO-185A.

[16] Aristotle Research Group, Georgia Institute of Technology, "TCAS," available at http://www.cc.gatech.edu/aristotle/Tools/subjects.

[17] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezzè, "Using symbolic execution for verifying safety-critical systems," in *ESEC / SIGSOFT FSE*, 2001, pp. 142–151.

[18] D. Detlefs, G. Nelson, and J. B. Saxe, "Simplify: a theorem prover for program checking." *J. ACM*, vol. 52, no. 3, pp. 365–473, 2005.

[19] K. L. McMillan, "An interpolating theorem prover," in *TACAS*, ser. LNCS, vol. 2988. Springer, 2004, pp. 16–30.

[20] G. J. Holzmann, "Software model checking with spin." *Advances in Computers*, vol. 65, pp. 78–109, 2005.

[21] K. McMillan, "Symbolic model checking: an approach to the state explosion problem," Ph.D. dissertation, Carnegie Mellon University, 1992, also available as CMU Technical Report CMU-CS-92-131.

APPENDIX

A. DEMONSTRATION DESCRIPTION

A.1 Abstraction and refinement

The EUREKA tool demonstration will first focus on the theoretical part of the work. The speaker will introduce the audience to the CounterExample-Guided Abstraction Refinement (CEGAR) approach, and will demonstrate the main advantages of the approach described in Section 2 with respect to the well-known CEGAR based on Boolean programs.

A.2 The input format

The demonstration will then draw the attention of the audience to the syntactic aspects of the input programs that will then be model-checked. The speaker will substantiate Linear Programs with arrays with a number of examples (see e.g. Figure 1) taken from the experiments shown in Tables 1, 2, and 3. Particularly, the speaker will focus on how to specify non-deterministic assignments and choices, and on the statements allowed by the syntax[1].

A.3 Using Eureka

The EUREKA tool is available as a binary executable for Linux systems. Thus it can be easily invoked from the shell prompt. The speaker will first explain the synopsis,

```
eureka [options] <file>,
```

Figure 2: Tool options: timings

and will then go through the possible `options` (see e.g. Figures 2 and 3) that allow the user to:

- **choose the refinement strategy:** the abstractor can expand all array indexes at once (option `-x`). This option is useful when the properties to verify involve all elements of an array. In doing so several abstraction and refinement steps are skipped, thus leading to a considerable leap in performance.
- **disable abstraction and refinement:** abstraction and refinement can be disabled when the input program is purely linear, that is, it does not contain accesses nor assignments to array elements (option `-l`).
- **enable debug mode:** option `-p` shows the abstract linear constraints computed by the Model Checking module for each node of the abstract control-flow graph. Option `-t` prints the control-flow graph and the transition relations associated to each of its nodes.
- **obtain timing information:** the user can be informed about the time taken by each module (abstraction, Model Checking, refinement) to accomplish its task (option `-t`).

The speaker will then explain the output of EUREKA when it is fed with a Linear Program with Arrays. Particularly, the speaker will focus on the output of the interprocedural analysis (either the program is safe or an abstract error trace is shown) and on the output of the refinement phase (either the abstract error trace is feasible or a number of array indexes need to be added to the initial abstraction). An example output is given in Figure 4.

[1]We recall that Linear Programs with arrays are in a fragment of the C programming language.

Figure 1: Example Linear Programs with arrays.

Figure 3: Tool invocation and synopsis

Finally, the user will remark the usefulness of our CEGAR approach and our implementation by comparing the results of EUREKA with the results obtained by tools such as SATABS and BLAST, as reported in Section 3.

B. TOOL AVAILABILITY

The EUREKA tool is implemented in C++ in order to take advantage of several libraries that efficiently handle arithmetic constraints and formulæ modulo theories. The (statically-linked) binary executable is currently available for Linux systems, and has been tested with the most recent Mandriva, Debian, and Ubuntu platforms.

The tool is currently available for beta-testing at `www.ai-lab.it/eureka/ase07`, together with a number of examples and programs used during the evaluation phase whose results are reported in Tables 1, 2, and 3.

Figure 4: The output of EUREKA on an instance of bubblesort.