SAFARI: SMT-based Abstraction For Arrays with Interpolants

Francesco Alberti[1], Roberto Bruttomesso[2], Silvio Ghilardi[2], Silvio Ranise[3], Natasha Sharygina[1]

[1] Formal Verification Lab, University of Lugano, Switzerland
[2] Università degli Studi di Milano, Milan, Italy
[3] FBK-Irst, Trento, Italy

Abstract. We present SAFARI, a model checker designed to prove (possibly universally quantified) safety properties of imperative programs with arrays of unknown length. SAFARI is based on an extension of lazy abstraction capable of handling existentially quantified formulæ for symbolically representing states. A heuristics, called term abstraction, favors the convergence of the tool by "tuning" interpolants and guessing additional quantified variables of invariants to prune the search space efficiently.

1 Introduction

Efficient and automatic static analysis of imperative programs is still an open challenge. A promising line of research investigates the use of model-checking coupled with abstraction-refinement techniques [2, 5, 8, 10, 14, 15] including Lazy Abstraction [3,12] and its later improvements that use interpolants during refinement [13]. An intrinsic limitation of the approaches based on Lazy Abstraction is that they manipulate quantifier-free formulæ to symbolically represent states. However, when defining properties over arrays, universal quantified formulæ are needed, e.g., as in specifying the property "the array is sorted". The tool we present, SAFARI, is based on a novel approach [1], in which Lazy Abstraction is used in combination with the backward reachability analysis behind the Model Checking Modulo Theories (MCMT) framework [9]. The resulting procedure allows checking safety properties for arrays that require universal quantification over the indices. Moreover, the presence of quantifiers requires particular care when computing interpolants. SAFARI comes with an efficient quantifier handling procedure, exploited to retrieve quantifier-free interpolation queries from instantiations of pairs of inconsistent quantified formulæ.

The paper presents the tool architecture and the implementation details such as heuristics for abstraction, interpolation tuning, quantifier handling, and synthesis of additional quantified variables in invariants.

Many efficient tools for imperative programs verification have been developed so far. The main difference between SAFARI and other model-checkers (e.g., Blast [3], Impact [13] and Magic [4]) is the ability of handling unbounded

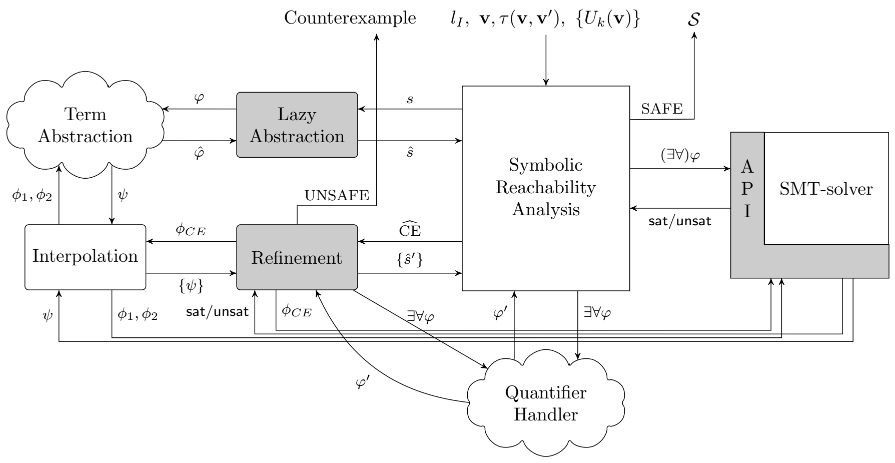

Fig. 1. Tool architecture.

arrays. Unlike ACSAR [14], SAFARI is able to discover new quantified predicates. Our approach does not require templates for predicate-discovering [16] and differs from abstract-interpretation techniques (e.g., [6, 7, 11]) in that it is based on a declarative framework that allows for identifying classes of array programs on which the core procedure terminates. For programs not meeting termination hypothesis user may suggest hints to SAFARI by means of accurate term abstraction lists as to help the tool to converge.

2 The Tool

The tool architecture is sketched in Fig. 1. SAFARI takes as input a transition system $(\mathbf{v}, \tau(\mathbf{v}, \mathbf{v}'))$ representing the encoding of an imperative program: $\mathbf{v}$ is the set of state variables among which some are arrays, and it always contains a variable `pc` ranging over a finite set $\{l_0, ..., l_n\}$ of program locations, among which we distinguish an initial location l_I.[1] A set of formulæ $\{U_k(\mathbf{v})\}$ representing unsafe states is also given to the tool; each U_k represents a violation of an assertion in the code. Next we describe the main modules of the tool.

Symbolic Reachability Analysis - This module implements a classical backward reachability analysis. Starting from the set of unsafe states, it repeatedly computes the pre-images with respect to the transition relation. It halts once it finds (the negation of a) *safe inductive invariant* $\mathcal{S}$ for the input system or when a run from the initial state to an unsafe state is found. The symbolic reachability search is based on the *safety* and the *covering* tests: the former checks the violation of an assertion while the latter implements the fix-point condition.

Lazy Abstraction - The search for a safe inductive invariant on the original (concrete) system may require a lot of resources or it cannot be computed because of possible divergence. To mitigate this problem, SAFARI relies on the

[1]The reader is referred to [1] for details.

Lazy Abstraction paradigm: in particular it extends it by allowing existentially quantified formulæ to represent states involving arrays. Moreover, SAFARI is able to introduce new quantified predicates on the fly, by means of *Term Abstraction* as described later on.

Quantifier Handling - The presence of quantified formulæ imposes particular attention during the satisfiability tests: available SMT-Solvers might not be able to automatically find suitable instances for the quantified variables. SAFARI provides a specific instantiation procedure, adapted from [9] to address this issue. To be effective, this procedure implements caching of information inside of specific data-structures used to represent formulæ. On one hand the caching increases the amount of space, on the other hand it cuts the number of instantiations due to constant-time checks. Alternatively, the quantified query may be passed to the SMT-Solver directly.

Refinement - This module receives an abstract counterexample and it checks first if the counterexample has a concrete counterpart. If so a feasible execution violating an assertion U_k is returned to the user. Otherwise the formulæ representing the states along the abstract execution trace have to be strengthened, possibly by adding new predicates, in order to rule out spurious executions. In the current implementation, refinement is performed by means of interpolation: the *Refinement* module iteratively interacts with the *Interpolation* module in order to retrieve quantifier-free interpolants.

Interpolation - Quantifier-free interpolation for formulæ involving arrays is in general not possible: in our case this situation is complicated by the presence of existential quantifiers. However, in [1], we show that the particular structure of the queries we handle, admits an equisatisfiable formulation at the quantifier-free level, for which meaningful quantifier-free interpolants can be computed. Quantifiers can be then reintroduced back, to preserve the original semantic of the formulæ. This technique, however, may not be sufficient to discover suitable new quantified predicates. To address this problem, SAFARI combines interpolation with a procedure called *Term Abstraction.*

Term Abstraction - Term Abstraction is a novel technique applied during the abstraction phase to select the "right" overapproximation to be computed, and during the refinement phase to "lift" the concrete infeasible counterexample to a more abstract level, by eliminating some terms. The effect of Term Abstraction is that of controlling both the abstraction function and the interpolants produced during refinement. Term Abstraction is discussed in detail in Section 3.

SMT-Solver - The tool relies on an SMT-Solver to decide satisfiability queries. An abstract interface provides an API to separate the actual SMT-Solver used and the services which are requested by SAFARI. This interface allows the invocation of different engines needed for particular tasks. SAFARI provides interface for OpenSMT and SMT-LIB v.2.

Implementation - SAFARI is written in C++ and can be downloaded from `http://verify.inf.usi.ch/safari`. Information on the usage of the tool and a full description of all the options can be found on the SAFARI's website.

3 Discussion

Term Abstraction and its benefits Term Abstraction is the main heuristic which distinguishes SAFARI from other tools based on abstraction-refinement. It works as follows. Suppose we are given an unsatisfiable formula of the form $\psi_1 \wedge \psi_2$, and a list of undesired terms $t_1, \ldots, t_n$ (called *term abstraction list*). The underlying idea is that terms in this list should be abstracted away for achieving convergence of the model checker. Iteratively we check if $\psi_1(c_i/t_i) \wedge \psi_2(d_i/t_i)$ is unsatisfiable, for c_i and d_i being fresh constants: if so then we set ψ_1 as $\psi_1(c_i/t_i)$ and ψ_2 as $\psi_2(d_i/t_i)$. Eventually we are left with an unsatisfiable formula $\psi_1 \wedge \psi_2$ where some undesired terms in $t_1, \ldots, t_n$ have been removed: the interpolant of ψ_1 and ψ_2, which can be computed with existing techniques, is likely to be free of the eliminated terms as well. SAFARI retrieves automatically from the input system a list of terms to be abstracted. The terms to abstract are usually set to iterators or variables representing the lengths of the arrays or the bounds of loops. The user can also suggest terms to be added to the list.

Synthesis of quantified invariants SAFARI is able to generate new quantified variables *if they are needed* to build the safe inductive invariant. Consider the pseudocode of Fig. 2: the first loop initializes all elements of the array a to 0, while the second loop sets a Boolean flag to false if a position with an uninitialized element is found. The program is clearly safe (the assertion is always satisfied), but since the length of the array a is not known, we need a quantified formula to represent the property Inv reached by every execution after the first loop. SAFARI is able to infer that formula automatically, even if the property to check does not contain any quantified variable (see line 9 of Fig.2 where the property involves the flag f only without any reference to the array a): this process of "synthesis" happens as a consequence of using of existentially quantified labels and term abstraction when refining a spurious (abstract) counterexample. The typical situation is that in which term abstraction succeeds in removing an iterator j from a concrete label of the form $\exists x.\ (j < n \wedge x = j \wedge a[x] \neq 0)$ to obtain $\exists x.\ (x < n \wedge a[x] \neq 0)$, where 0 can be any other constant depending on the example. The new label contains no reference to the original iterator j, it is more abstract, and it resembles the structure of Inv (once negated: recall that our approach is backward). In short, term abstraction is used during refinement to lift an infeasible concrete trace (corresponding to a spurious abstract counterexample) to the most abstract level with respect to a set of terms. As a side effect, a quantified predicate may be inferred.

```
1  $i = 0$;
2  while( $i < n$ )
3     $a[i] = 0$;
4     $i = i + 1$;
$\boxed{\text{Inv: } \forall x.\ (0 \leq x < n) \Rightarrow a[x] = 0}$
5  $j = 0$; $f =$ true;
6  while( $j < n$ )
7     if( $a[j]! = 0$ ) $f =$ false;
8     $j = j + 1$;
9  assert ( $f$ );
```

Fig. 2. Pseudo-code for "init and test".

Quantifier handling The approach behind SAFARI relies on Lazy Abstraction combined with the MCMT framework [1]. Intuitively, during the backward-reachability from the set of error states, we keep track of the array index positions of interest (the positions that are accessed for read) with existentially quantified variables. Safety and covering checks can be performed with dedicated instantiation heuristics. Whereas safety tests are decidable [1], covering tests must be dealt with incomplete algorithms based on clever instantiations (incompleteness of covering tests do not affect the soundness of the tool, they can only affect termination chances). In addition, special care is needed when discovering new predicates via interpolation: quantified queries (expressing trace feasibility) can be Skolemized and instantiated, thus producing equisatisfiable quantifier-free queries. These quantifier-free queries belong to a fragment of the theory of arrays enjoying quantifier-free interpolation. Then, quantifier-free interpolants are computed to refine node labeling, where existential quantifiers are re-introduced by existentially quantifying the Skolem constants (see [1] for details).

4 Experiments

We applied SAFARI to the verification of various problems with arrays. None of these problems can be solved by SAFARI without abstraction. Table 1 reports some experimental results (obtained running SAFARI on an Intel i7 @2.66 GHz, 4GB of RAM running OSX 10.7). More statistics can be found on SAFARI website. The benchmarks have been run with the most efficient options, namely with "Term Abstraction" both for abstraction and refinement. The term abstraction list is automatically computed for all the benchmarks but those marked with a star: in those few cases a user-defined list has been provided. The table reports variations of the same problems (v1,v2) and properties specified (P1,P2). The benchmarks marked "buggy" were injected with a bug that invalidates the property. To test the flexibility of SAFARI, we also verified some randomly generated problems taken from those shipped with the distribution of the ARMC model-checker (`http://www.mpi-sws.org/~rybal/armc/`). They consists of safety properties of numerical programs without arrays. For those, our tool can solve 23 out of 28 benchmarks with abstraction, but only 9 without using it. For all of these problems, SAFARI automatically retrieves a suitable term abstraction list. For those benchmarks that could be solved even without abstraction, the overhead of abstraction is generally negligible.

Acknowledgements The work of the first author was supported by the Hasler Foundation under project 09047 and that of the fourth author was partially supported by the "SIAM" project founded by Provincia Autonoma di Trento in the context of the "team 2009 - Incoming" COFUND action of the European Commission (FP7).

References

1. F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. Lazy Abstraction with Interpolants for Arrays. In *LPAR-18*, pages 46–61, 2012.
2. T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic Predicate Abstraction of C Programs. In *PLDI*, pages 203–213, 2001.

Benchmark	Time (s)	SMT-calls	Iter.	P. vars	$\|\mathcal{S}\|$	$\mathcal{S}$ vars	Status
binary sort*	0.3	817	2	2	21	4	SAFE
filter (P1)	0.03	27	0	1	2	1	SAFE
filter (P2)	0.04	28	0	1	2	1	SAFE
filter (all)	0.03	34	0	1	3	1	SAFE
find (v1, P1)	0.6	171	3	1	7	5	SAFE
find (v1, P1, buggy)	0.05	71	1	1	-	-	UNSAFE
find (v1, P2)	0.06	65	1	1	4	3	SAFE
find (v1, all)	0.8	246	4	1	12	5	SAFE
find (v2)	0.08	50	1	1	3	1	SAFE
init and test	0.3	352	3	0	13	2	SAFE
initialization	0.1	90	1	1	4	4	SAFE
integers	0.02	20	0	0	2	0	SAFE
max in array	0.9	1237	8	1	29	3	SAFE
max in array (buggy)	0.1	235	2	1	-	-	UNSAFE
partition (v1, P1)	0.05	32	0	1	4	1	SAFE
partition (v1, P2)	0.06	32	0	1	4	1	SAFE
partition (v1, all)	0.08	63	0	1	4	1	SAFE
partition (v2)	0.04	33	0	1	2	2	SAFE
partition (v2, buggy)	0.09	62	0	1	-	-	UNSAFE
selection sort*	0.6	478	4	2	15	3	SAFE
selection sort (buggy)	1.9	1957	8	2	-	-	UNSAFE
strcmp	0.2	308	4	1	12	2	SAFE
strcpy	0.02	16	0	1	2	2	SAFE
vararg*	0.05	48	0	1	3	2	SAFE

Table 1. Total time, calls to SMT, CEGAR iterations, quantified variables in the assertion, size of the covering set, quantified variables in the covering set.

3. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast. *STTT*, 9(5-6):505–525, 2007.
4. Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in c. In *ICSE*, pages 385–395, 2003.
5. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *CAV*, pages 154–169, 2000.
6. P. Cousot, R. Cousot, and F. Logozzo. A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis. In *POPL*, 2011.
7. Isil Dillig, Thomas Dillig, and Alex Aiken. Fluid updates: Beyond strong vs. weak updates. In *ESOP*, pages 246–266, 2010.
8. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL*, pages 191–202, 2002.
9. S. Ghilardi and S. Ranise. MCMT: A Model Checker Modulo Theories. In *IJCAR*, pages 22–29, 2010.
10. S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *CAV*, pages 72–83, 1997.
11. N. Halbwachs and Mathias P. Discovering Properties about Arrays in Simple Programs. In *PLDI'08*, pages 339–348, 2008.
12. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *POPL*, pages 58–70, 2002.
13. K. L. McMillan. Lazy Abstraction with Interpolants. In *CAV*, 2006.
14. M. N. Seghir, A. Podelski, and T. Wies. Abstraction Refinement for Quantified Array Assertions. In *SAS*, pages 3–18, 2009.
15. S.Lahiri and R. Bryant. Predicate Abstraction with Indexed Predicates. *TOCL*, 9(1), 2007.
16. S. Srivastava and S. Gulwani. Program Verification using Templates over Predicate Abstraction. In *PLDI*, 2009.