Booster: an acceleration-based verification framework for array programs*

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

[1] University of Lugano, Lugano, Switzerland
[2] Università degli Studi di Milano, Milan, Italy
[3] VERIMAG, Grenoble, France

Abstract. We present Booster, a new framework developed for verifiying programs handling arrays. Booster integrates new acceleration features with standard verification techniques, like Lazy Abstraction with Interpolants (extended to arrays). The new acceleration features are the key for scaling-up in the verification of programs with arrays, allowing Booster to efficiently generate required quantified safe inductive invariants attesting the safety of the input code.

1 Introduction

In this paper we present Booster, a tool for the verification of software systems handling arrays. The novelty of Booster with respect to other tools supporting array analysis [7,10,11,13,14,17] is its being based on *acceleration* procedures.

Acceleration procedures target the generation of the transitive closure of relations encoding system evolution. In our case, acceleration is applied to relations encoding loops of the analyzed program. With respect to abstraction-based procedures, acceleration offers a *precise* solution (not involving over-approximations) to the problem of computing the reachable state-space of a transition system, but on the other side has syntactic restrictions preventing its general application. On the other side, abstraction-based solutions are usually a very general framework, but they also require heuristics (and in some cases even user guidance) in order to increase their practical effectiveness. As an example, the *Lazy Abstraction with Interpolants* one (lawi [3,18]), which has been shown to be one of the most effective abstraction-based framework in verification [8], relies on Craig interpolants for refining the level of abstraction. Craig interpolants, however, are not unique, and it has been shown that different interpolants might seriously affects the performance of the verification task [19].

Booster exploits acceleration in two different ways. Accelerations of loops falling in decidable fragments are handled precisely, following the schema presented in [6]. Those requiring over-approximations and suitable refinement procedures (as discussed in [5]) are handled by an improved version of the mcmt model-checker [15], the fixpoint engine integrated in Booster.

*The work of the first author was supported by Swiss National Science Foundation under grant no. P1TIP2_152261.

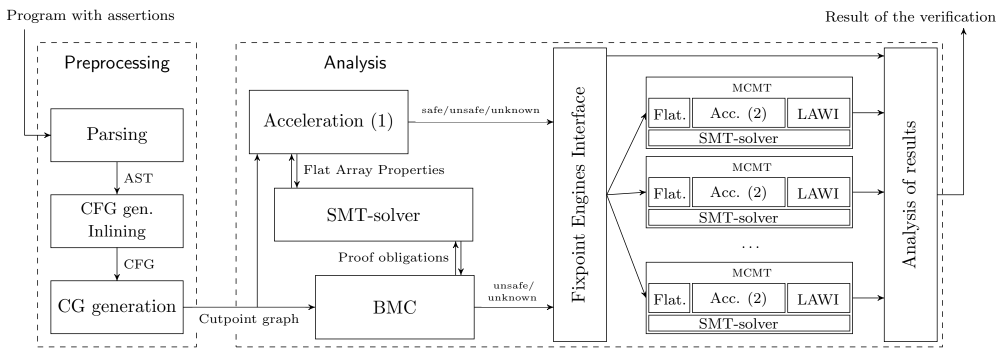


Fig. 1: The architecture of Booster.

The architecture of Booster, detailed in the next section, is structured according to the standard compilers architecture, where the initial parsing phase generates an intermediate representation of the code which is subject to several optimizations before being fed to an engine for checking its safety. From this point of view, acceleration can be viewed as the most important and distinguishing optimization of our approach, while an abstraction-based module acts as the engine performing the analysis.

Our experimental evaluation, performed on a benchmark suite comprising programs with arrays selected from heterogeneous sources, attests the effectiveness of our new tool and the impressive benefits brought by acceleration procedures.

2 The Tool

Booster is written in C++, and it is available at `http://www.inf.usi.ch/phd/alberti/prj/booster/`. Fig. 1 depicts its architecture. In this section we describe the features implemented in Booster.

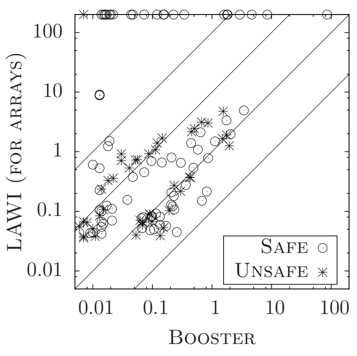
Preprocessing. Given a program, Booster generates its control-flow graph (CFG) and inlines procedure calls. From the CFG, Booster builds the *cutpoint graph* (CG) of the input program [16]. A cutpoint graph is a graph-representation of the input code where each vertex represents either the entry/exit block of the program or a loop-head, and the edges are labeled with sequences of assumptions or assignments. The representation of the input code as a cutpoint graph is extremely beneficial for applying acceleration techniques, and it is adopted to maximize the application of acceleration procedures. Indeed, acceleration techniques for code handling arrays can be applied only to transitions representing self-loops (and matching some other syntactic patterns [5, 6]).

BMC. This module has been devised as a preliminary rather rough analysis: Booster adopts a Bounded Model Checking approach [9] at the very beginning of the analysis in order to detect unsafe programs *before* enabling analysis (like

acceleration) with a high impact on the tool performances[4]. A low number of unwindings constitutes, at this stage of the analysis, a good trade-off between precision (number of unsafe programs detected) and efficiency.
Acceleration (1). This module targets the verification of $\mathsf{simple}^0_{\mathcal{A}}$-programs [6]. These kind of programs are characterized by (i) having a flat control-flow structure, i.e., each location belongs to at most one loop, and (ii) comprising only loops that can be accelerated as a "Flat Array Properties", i.e., $\exists\forall$-formulæ of the theory of arrays admitting a decision procedure for checking their (un)satisfiability. If the given CG is a $\mathsf{simple}^0_{\mathcal{A}}$-program, Booster accelerates all the loops. This is a cheap template-based pattern matching task: being a $\mathsf{simple}^0_{\mathcal{A}}$-program, all the loops of the program match the pattern given in [6]. The loops are substituted with their accelerated counterparts; subsequently Booster generates the proof-obligations, which are Flat Array Properties, required to check the (un)safety of the program. Unfortunately, this fragment is not entirely covered by decision procedures implemented in available SMT-solvers. In practice, Booster relies on the Z3 SMT-solver [12] for solving such queries. The SMT-solver is usually very efficient on unsatisfiable proof obligations, but might struggle on satisfiable ones. The BMC analysis executed before this module, however, is generally able to find the corresponding traces, reporting the unsafety of the code before starting this acceleration procedure. It is also important to notice that, at this stage of the analysis, Booster exploits the *full power* of acceleration on a well-defined class of transitions, i.e., the loops of $\mathsf{simple}^0_{\mathcal{A}}$-programs.
Transition System generation. If the program is not a $\mathsf{simple}^0_{\mathcal{A}}$-program or the SMT-solver exploited by the "Acceleration (1)" module times out, the CG of the program is translated into a transition system and then fed into mcmt.

MCMTv2.5 mcmt is a model-checker based on a backward reachability analysis approach for array-based transition systems, formal models suitable for the representation of many classes of real systems, including programs with arrays. mcmt is written in C and available at `http://users.mat.unimi.it/users/ghilardi/mcmt/`. The version of mcmt included in Booster extends the previous version [15] implementing (i) the new Lazy Abstraction with Interpolants (LAWI) for Arrays approach [1] and (ii) acceleration procedures for array relations [5] (this also differentiates mcmt from safari [2]).
Flattening. Flattening is a preprocessing technique exploited inside mcmt to reduce the transition formulæ and state formulæ to a format where array variables are indexed only by existentially quantified variables. It is based on the rewriting rule $\phi(a[t], ...) \rightsquigarrow \exists x(x = t \wedge \phi(a[x], ...))$. This format is particularly indicated for inferring *quantified* predicates within the lawi framework and it is exploited by the *term abstraction* heuristic [2].
Acceleration (2). mcmt adopts acceleration as a preprocessing step, following the approach described in [5]. In contrast with the "Acceleration (1)" module dis-

[4]Formulæ generated by the "Acceleration (1)" module contain alternation of quantifiers and it has been proven that checking their satisfiability may be a NExpTime-complete problem [6].



(a)

BENCHMARK	STATUS	TIME (s)
set property	SAFE	1.60
set property (bug)	UNSAFE	1.95
bubble sort	SAFE	0.23
bubble sort (bug)	UNSAFE	0.09
palindrome	SAFE	0.02
sentinel	SAFE	0.01
strcpy	SAFE	0.01
strcmp	SAFE	0.02
init even	SAFE	0.02
double swap	SAFE	0.16
merge interleave	SAFE	0.09
merge interleave (bug)	UNSAFE	0.11

(b)

Fig. 2: BOOSTER performances.

cussed previously, acceleration here is applied to a wider class of transitions, but preimages along accelerated formulæ are not kept precise given their intractable format[5], but are over-approximated with their *monotonic abstraction* [4].
LAWI. MCMT implements the Lazy Abstraction with Interpolants for Array framework (following the description given in [2]) enhanced with a suitable refinement procedure for handling the over-approximations introduced to exploit accelerated relations [5].

Portfolio approach The "term abstraction" heuristic has a great impact on the performances of the LAWI framework for arrays [2]. It leverages the flat encoding of formulæ manipulated by the model-checker in order to generate quantified predicates for a successful array analysis. BOOSTER nullifies the required user ingenuity for defining a proper term abstraction list. Internal heuristics, inherited from [2], generate some suitable term abstraction lists. The fixpoint engine is subsequently executed adopting a portfolio approach, according to which BOOSTER generates several parallel instances of MCMT, each with different settings (including different term abstraction lists).

3 Experimental evaluation and Conclusion

We evaluated BOOSTER on a large set of programs (both safe and unsafe) with arrays taken from several heterogeneous sources. Fig. 2a compares BOOSTER running time with and without acceleration procedures[6]. This figure clearly shows that acceleration is a key feature in the BOOSTER framework: it significantly reduces the divergence cases and allows to achieve a speed-up up to two orders of magnitude. We also report that the (un)safety of many programs (roughly

[5]These ∃∀-formulæ might produce proof obligations falling outside known decidable fragments of array theories and may invalidate the internal heuristics of MCMT.

[6]Without acceleration the verification is performed entirely by the LAWI module.

the 50% of our benchmark suite) is detected directly by the "BMC" and "Acceleration (1)" modules, remarking the importance of acceleration in a software verification framework. We report in Table 2b some statistics about BOOSTER running times for challenging well-known benchmarks in array-analysis literature, observing that, to the best of our knowledge, there are no tools able to deal with all the programs in our benchmark suite.

References

1. F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. Lazy abstraction with interpolants for arrays. In *LPAR*, pages 46–61, 2012.
2. F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. SAFARI: SMT-Based Abstraction for Arrays with Interpolants. In *CAV*, pages 679–685, 2012.
3. F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. An extension of lazy abstraction with interpolation for programs with arrays. *FMSD*, 2014. To appear.
4. F. Alberti, S. Ghilardi, E. Pagani, S. Ranise, and G.P. Rossi. Universal guards, relativization of quantifiers, and failure models in model checking modulo theories. *JSAT*, 8(1/2):29–61, 2012.
5. F. Alberti, S. Ghilardi, and N. Sharygina. Definability of accelerated relations in a theory of arrays and its applications. In *FroCoS*, pages 23–39, 2013.
6. F. Alberti, S. Ghilardi, and N. Sharygina. Decision procedures for flat array properties. In *TACAS*, pages 15–30, 2014.
7. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. VeriMAP: A tool for verifying programs through transformations. In *TACAS*, pages 568–574, 2014.
8. D. Beyer. Status report on software verification - (competition summary sv-comp 2014). In *TACAS*, pages 373–388, 2014.
9. A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *TACAS*, pages 193–207, 1999.
10. N. Bjørner, K. McMillan, and A. Rybalchenko. On solving universally quantified Horn clauses. In *SAS*, pages 105–125, 2013.
11. P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL*, pages 105–118, 2011.
12. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.
13. I. Dragan and L. Kovács. LINGVA: Generating and proving program properties using symbol elimination. In *PSI*, 2014. To appear.
14. P. Garg, C. Löding, P. Madhusudan, and D. Neider. Ice: A robust framework for learning invariants. In *CAV*, pages 69–87, 2014.
15. S. Ghilardi and S. Ranise. MCMT: A Model Checker Modulo Theories. In *IJCAR*, pages 22–29, 2010.
16. A. Gurfinkel, S. Chaki, and S. Sapra. Efficient predicate abstraction of program summaries. In *NASA Formal Methods*, pages 131–145, 2011.
17. K. Hoder, L. Kovács, and A. Voronkov. Invariant Generation in Vampire. In *TACAS*, pages 60–64, 2011.
18. K.L. McMillan. Lazy abstraction with interpolants. In *CAV*, pages 123–136, 2006.
19. S.F. Rollini, L. Alt, G. Fedyukovich, A.E.J. Hyvärinen, and N. Sharygina. PeRIPLO: A framework for producing effective interpolants in sat-based software verification. In *LPAR*, pages 683–693, 2013.