A framework for the verification of parameterized infinite-state systems*

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 our tool, developed for the analysis and verification of parameterized infinite-state systems. The framework has been successfully applied in the verification of programs handling unbounded data-structures. In such application domain, being able to infer quantified invariants is a mandatory requirement for successful results. We will describe the techniques implemented in our system and discuss how they contribute in achieving important results in the analysis of parameterized distributed and timed systems, as well as of programs with arrays of unknown length.

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 [11, 20, 27, 30, 35, 36] including Lazy Abstraction [12, 32] and its later improvements that use interpolants during refinement [34]. 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 MCMT (in the new version 2.5) is based on a novel approach [4], in which Lazy Abstraction is used in combination with the backward reachability analysis of array-based systems [28]; recent acceleration techniques for arrays [8, 9] have also been significantly (although not yet completely) included in the latest version of the tool.

2 The Tool

MCMT takes as input a transition system $(\mathbf{v}, \tau(\mathbf{v}, \mathbf{v}'), \iota(\mathbf{v}))$ representing the encoding of an array-based system [28]: this can be a parameterized distributed system, a network of timed automata, an imperative program, etc. The essential nature of the specification system is its *parametricity*: a finite (but unspecified)

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

number of components takes part in it, the components being interpreted as single processes, agents, array cells, etc. Formally, the array based system is specified by fixing a tuple $\mathbf{v}$ of state variables, a formula $\iota(\mathbf{v})$ describing initial states and a formula $\tau(\mathbf{v}, \mathbf{v}')$ relating current variables $\mathbf{v}$ with their updated counterparts $\mathbf{v}'$. State variables are typed and some of them represent arrays, modeled as free function symbols from a sort of indexes `INDEX` to some elements $\texttt{ELEM}_1, \ldots, \texttt{ELEM}_k$ sorts. The type of indexes is natural numbers, whereas the types of elements can be integers, Boolean, reals, enumerated data-types, etc. A set of formulæ $\{U_k(\mathbf{v})\}$ representing unsafe states is also given to the tool; each U_k represents an undesired property, e.g. a violation of an assertion in the code. Next we describe the main features of the tool.

Symbolic Reachability Analysis - This module implements a classical backward reachability analysis [1–3]. 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 an 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 tests fix-points.

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, MCMT extends the Lazy Abstraction paradigm by allowing existentially quantified formulæ to represent states. Moreover, MCMT is able to introduce new quantified predicates on the fly, by means of *Term Abstraction* or *Acceleration*, see below.

Acceleration - Acceleration is a well established technique in model-checking: the acceleration (i.e. the transitive closure) of a relation encoding systems evolution (like loops in programs) allows us to compute 'in one shot' the reachable set of states after an arbitrary but finite number of execution steps. This has the great advantage of keeping under control sources of (possible) divergence arising in the reachability analysis. Definability results for accelerations are well known in numerical domains like difference bounds constraints [17, 21], octagons [14] and finite monoid affine transformations [26] (the paper [16] presents a general approach covering all these domains); however, little is known for more complex data structures like arrays (but see [15]). In [8,9] it is shown that the acceleration of relations corresponding to some classes of guarded assignments over arrays lead to formulæ in decidable fragments of the theory of arrays; as a consequence, some common classes of imperative programs over arrays (including those implementing searching, initializing, finding, copying, comparing functions) have decidable reachability problems. Acceleration for arrays is another way of introducing quantifiers in formulæ describing reachable states; it is only partially implemented in MCMT, where it is exploited for over-approximations in abstraction/refinements loops.

Quantifier Handling - The presence of quantified formulæ imposes particular attention during the satisfiability tests: available SMT-Solvers might not be able to deal automatically with such quantified formulæ. MCMT provides a

specific instantiation procedure, adapted from [29] 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.

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 corresponding to some satisfied 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 a form of interpolation guided by term abstraction.

Term Abstraction - Term Abstraction [4] is a novel technique applied during the abstraction phase to select the "right" over-approximation to be computed, and during the refinement phase to "lift" the concrete infeasible counterexample to a more abstract level, by eliminating some terms. Term Abstraction (implemented also in the SAFARI tool [5]) is the main heuristic which distinguishes MCMT from other tools based on abstraction-refinement. It works as follows. Suppose we are given 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, these terms are abstracted out (if possible) from formulæ over-approximating sets of reachable states; one way to do this is to replace them by fresh free constants, so that they are likely not to occur anymore in interpolants or in formulæ to which quantifier elimination is applied. MCMT retrieves automatically from the input system a list of terms to be abstracted. The terms to be abstracted 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.

Specification Syntax - MCMT has its own specification language (roughly, an extension of the specification language of the underlying SMT-Solver YICES). Even though it has been significantly improved, it is still rather low level. Such a language is exploited by the BOOSTER verifying compiler[4].

3 Implementation and Related Work

MCMT is written in C and can be downloaded from `http://users.mat.unimi.it/users/ghilardi/mcmt/`. Information on the usage of the tool and a full description of all the options can be found on the User Manual that can be downloaded from MCMT's website. The use of the appropriate options is crucial not only for performances, but also for convergence (some benchmarks can be solved by plain backward search, in some cases run-time invariant search can be exploited to speed up the tool, for imperative programs it is essential to use abstraction/refinement mode or acceleration or both). MCMT relies on

[4]The interested reader is pointed to the BOOSTER web-page, `http://inf.usi.ch/phd/alberti/prj/booster` for more information about it.

the SMT-Solver Yices (see `http://yices.csl.sri.com/`) to decide satisfiability queries; the solver can be linked to the model-checker on a client/server architecture via API. MCMT distribution includes about 100 examples files taken from different sources (cache coherence and mutual exclusion problems, timed and fault tolerant systems, imperative programs, etc); more examples related to specific case studies [6, 7, 18, 19] can be reached from MCMT web-page. A Table covering few experimental results is attached in the Appendix below.

Current literature on infinite state model checking is extremely large, however it is much more limited if we restrict to papers and tools handling parametric specifications. For distributed systems, the best performing tool (resulting from an extension and a re-implementation on a parallel architecture of MCMT framework) is probably CUBICLE [22, 23]. In software model checking, unbounded arrays problems have been attacked from various viewpoints, including abstract interpretation [24, 25, 31], program transformations [10], predicate abstraction [35, 36] and template/constraints generation [13, 33, 37]. An experimental tool comparison is difficult for various practical reasons; however, since most benchmarks are taken from common sources, from the results reported in the above mentioned papers, it seems that MCMT compares well, both in terms of performances and in terms of solved instances.

References

1. P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *Proc. of LICS*, pages 313–321, 1996.
2. P. A. Abdulla, G. Delzanno, N. B. Henda, and A. Rezine. Regular model checking without transducers. In *TACAS*, volume 4424 of *LNCS*, pages 721–736, 2007.
3. P. A. Abdulla, G. Delzanno, and A. Rezine. Parameterized verification of infinite-state processes with global conditions. In *CAV*, pages 145–157, 2007.
4. F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. Lazy Abstraction with Interpolants for Arrays. In *LPAR-18*, pages 46–61, 2012.
5. 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.
6. F. Alberti, S. Ghilardi, E. Pagani, S. Ranise, and G. P. Rossi. Automated support for the design and validation of fault tolerant parameterized systems - a case study. In *Proc. of AVOCS*, 2010.
7. F. Alberti, S. Ghilardi, E. Pagani, S. Ranise, and G. P. Rossi. Brief announcement: Automated support for the design and validation of fault tolerant parameterized systems - a case study. In *DISC*, pages 392–394, 2010.
8. 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.
9. F. Alberti, S. Ghilardi, and N. Sharygina. Decision procedures for flat array properties. In *TACAS*, pages 15–30, 2014.
10. E. De Angelis, F. Fioravanti, M. Proietti, and A. Pettorossi. Verifying Array Programs by Transforming Verification Conditions. In *VMCAI*, 2014.
11. T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic Predicate Abstraction of C Programs. In *PLDI*, pages 203–213, 2001.
12. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast. *STTT*, 9(5-6):505–525, 2007.

13. N. Björner, K. McMillan, and A. Rybalchenko. On solving universally quantified Horn clauses. In *SAS*, pages 105–125, 2013.
14. M. Bozga, C. Girlea, and R. Iosif. Iterating octagons. In *TACAS*, LNCS, pages 337–351, 2009.
15. M. Bozga, P. Habermehl, R. Iosif, F. Konecný, and T. Vojnar. Automatic verification of integer array programs. In *CAV*, pages 157–172, 2009.
16. M. Bozga, R. Iosif, and F. Konecny. Fast acceleration of ultimately periodic relations. In *CAV*, LNCS, 2010.
17. M. Bozga, R. Iosif, and Y. Lakhnech. Flat parametric counter automata. *Fundamenta Informaticae*, (91):275–303, 2009.
18. R. Bruttomesso, A. Carioni, S. Ghilardi, and S. Ranise. Automated Analysis of Parametric Timing Based Mutual Exclusion Protocols. In *NASA Formal Methods Symposium*, 2012.
19. A. Carioni, S. Ghilardi, and S. Ranise. MCMT in the land of parameterized timed automata. In *In proc. of VERIFY*, 2010.
20. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *CAV*, pages 154–169, 2000.
21. H. Comon and Y. Jurski. Multiple counters automata, safety analysis and presburger arithmetic. In *CAV*, volume 1427 of *LNCS*, pages 268–279. Springer, 1998.
22. S. Conchon, A. Goel, S. Krsti, A. Mebsout, and F. Zadi. Cubicle: a Parallel SMT-based Model-Checker fro Parameterized Systems. In *Proc. of CAV*, LNCS, 2012.
23. S. Conchon, A. Goel, S. Krsti, A. Mebsout, and F. Zadi. Invariants for Finite Instances and Beyond. In *Proc. of FMCAD*, 2013.
24. P. Cousot, R. Cousot, and F. Logozzo. A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis. In *POPL*, 2011.
25. I. Dillig, T. Dillig, and A. Aiken. Fluid Updates: Beyond Strong vs. Weak Updates. In *Programming Languages and Systems*. 2010.
26. A. Finkel and J. Leroux. How to compose Presburger-accelerations: Applications to broadcast protocols. In *FST TCS 02*, pages 145–156. Springer, 2002.
27. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL*, pages 191–202, 2002.
28. S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Towards SMT Model-Checking of Array-based Systems. In *Proc. of IJCAR*, LNCS, 2008.
29. S. Ghilardi and S. Ranise. MCMT: A Model Checker Modulo Theories. In *IJCAR*, pages 22–29, 2010.
30. S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *CAV*, pages 72–83, 1997.
31. N. Halbwachs and Mathias P. Discovering Properties about Arrays in Simple Programs. In *PLDI'08*, pages 339–348, 2008.
32. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *POPL*, pages 58–70, 2002.
33. D. Larraz, E. Rodríguez-Carbonell, and A. Rubio. SMT-based array invariant generation. In *VMCAI*, pages 169–188, 2013.
34. K. L. McMillan. Lazy Abstraction with Interpolants. In *CAV*, 2006.
35. M. N. Seghir, A. Podelski, and T. Wies. Abstraction Refinement for Quantified Array Assertions. In *SAS*, pages 3–18, 2009.
36. S.Lahiri and R. Bryant. Predicate Abstraction with Indexed Predicates. *TOCL*, 9(1), 2007.
37. S. Srivastava and S. Gulwani. Program Verification using Templates over Predicate Abstraction. In *PLDI*, 2009.

A Some experimental data

We report in the Table below some experimental data on benchmark problems; we made a selection including both easy well known problems and more challenging ones. The experiments were run on a laptop Intel(R) Core(TM) i3 CPU 2.27GHz with 4GB RAM running Linux Ubuntu 12.04.

In the second column we indicate the class of the problem: (M) mutual exclusion, (C) cache coherence, (D) other distributed protocols (timed, fault tolerant, etc.), (S) sequential program for arrays, (S+) sequential program for arrays with nested loops. In column 3-8 we respectively give the depth of the search tree, the number of nodes generated by the tool in the search tree, the number of subsumed or subcovered nodes, the number of calls to the SMT solver, the number of invariants found by the tool in forward search and the number of refinements applied in abstraction/refinement mode. In the last column we put the total time in seconds and in the last-but-one column the options used (A=acceleration, AR=abstraction/refinement, I=invariant search).[5] For each problem we reported the result in the best configuration we found for the tool.

Problem	kind	d	#n	#del	#SMT	#inv	#ref	heur	time
Illinois	(C)	4	8	0	212	0	0	-	0.06
German	(C)	26	2121	255	117121	0	0	-	60.76
German_buggy	(C)	16	1300	203	24275	0	0	-	14.28
Bakery	(M)	2	1	0	29	0	0	-	0.00
Szymanski	(M)	11	17	5	1092	12	0	I	0.21
Szymanski_atomic	(M)	19	63	7	5470	32	0	I	1.82
Distributed Lamport	(M)	23	248	42	19622	7	0	I	27.18
Crash	(D)	13	113	21	1731	0	0	-	0.75
Fischer	(D)	10	16	2	363	0	0	-	0.08
Fischer_buggy	(D)	6	16	0	307	0	0	-	0.06
Lynch-Shavit_full	(D)	25	1103	99	56638	0	0	-	33.39
Strcpy	(S)	4	4	2	48	0	0	A	0.01
Strcmp	(S)	6	10	4	128	0	0	A	0.02
Max_in_array	(S)	7	13	6	166	0	0	A	0.04
Reverse	(S)	4	8	5	101	0	0	A	0.03
Palindrome	(S)	4	7	4	107	0	0	A	0.04
AllDifferent	(S+)	7	49	39	871	0	8	A + AR	0.40
BubbleSort	(S+)	5	14	10	200	0	0	A	0.07
InsertionSort	(S+)	18	98	56	3874	0	2	AR	1.43
SelectionSort	(S+)	8	101	77	6059	8	11	AR + I	4.98

[5] Corresponding command line options: `-Z` gives acceeration, `-i1, -i2, -i3, -a, -I` give different invariant searches in normal mode, `-AN` gives abstraction-refinement (with max `N` refinements per node), `-CN` gives abstraction-refinement (with max `N` refinements per node) together with a specific form of invariant search.