Efficient Predicate Analysis of MISRA-C Programs

Feng Gao, Li Li, Jie Luo *

State Key Lab of Software Development Environment

Department of Computer Science and Engineering, Beihang University

Beijing 100191, China

{gaofeng,lili,luojie}@nlsde.buaa.edu.cn

Keywords: MISRA-C; Predicate Abstraction; Lazy Abstraction; CEGAR; Imperative Predicates

Abstract. Great care needs to be exercised when using C within safety-related systems. MISRA-C defines a suitable subset of C to be used in safety-related software development, which is easier for program analysis. Predicate abstraction refinement is one of the leading approaches to software verification. In this paper, we propose a procedure to analyze MISRA-C program with predicate abstraction efficiently. The efficiency of this process depends on lazy abstraction and imperative predicates set, which are designed for the *program abstraction* and *predicate refinement* procedures respectively. Besides, some features have been added to obtain the desired efficiency, such as initial predicates, pointer alias analysis and so on. Experiments show that it can result in a significant reduction of analysis time and improvement of memory usage compared to earlier methods.

Introduction

Several important producers of safety critical systems, in particular in the field of avionics and nuclear power plant systems, develop software using subsets of the C language to avoid linguistic features, such as dynamic allocations of memory, that are likely to cause failures difficult to detect during verification. MISRA-C is such a guideline that defines a suitable subset of C to be used in safety-related software development [1]. With a subset of C excluding these difficult aspects, some static analysis methods will perform better, for example, predicate abstraction (model checking).

It is widely believed that effective model checking [2] of software systems could produce major enhancements in software reliability and robustness. However, the effectiveness of model checking of such systems is severely constrained by the state space explosion problem. *Predicate abstraction* [3, 4] is one of the most popular and widely applied methods for systematic abstraction of programs. Roughly, Abstraction consists of constructing an abstract program P^a from a given program P in such a way that the set of possible executions of P is a subset of those of P^a; the vice-versa does not hold. The abstraction refinement process has been automated by the *Counterexample Guided Abstraction Refinement* paradigm [5, 6], or CEGAR for short. One of the most difficult problems in CEGAR is to identify, during the refinement phase, appropriate criteria to discover new predicates that provide better abstractions. In this respect, *Lazy Abstraction* is particularly interesting since it is capable of refining the abstraction by using different degrees of precision for different parts of the program. The idea is to use a control-flow graph to keep track of how the program locations are traversed and of predicates to represent the data-flow and the program annotations.

Given a set of predicates P, the process of constructing abstraction is in the worst case exponential, both in time and space, in $|P|$. Therefore, a crucial point in deriving efficient algorithms based on predicate abstraction is the choice of a small set of predicates. In previous work [7] the refinement is done by adding predicates that eliminate the new spurious counterexample while maintaining the predicates that were found in previous iterations. However, this accumulative approach cannot guarantee an imperative set of predicates, because it depends on the order in which the counterexamples are identified and the choice of predicates at each step.

In this paper, we described a technique for improving the performance of the abstract refinement loop by applying lazy abstraction with an imperative predicates set, which is expected to reduce the

© 2015. The authors - Published by Atlantis Press

overall verification time and required space. Our experimental results show that indeed the number of predicates and consequently the amount of memory required are significantly reduced.

Related Work

Abstraction techniques are often based on the abstract interpretation work of Cousot and Cousot [8] and require the user to give an abstraction function relating concrete datatypes to abstract datatypes. The notion of CEGAR was originally introduced by Kurshan [9] (originally termed localization) for model checking of finite state models. CEGAR for ANSI-C programs was promoted by the success of the Slam project at Microsoft [10]. Thus, there are already a number of other implementations, such as Magic [11] and Blast [12].

The BLAST toolkit [12] introduced the notion of lazy abstraction, where the abstraction refinement is completely demand-driven to remove spurious behaviors. The abstraction is constructed on-the-fly and only to the required precision. Yogi [13] is a tool that implements the Dash algorithm, which combines testing and verification in order to achieve better performance. Strichman et al. [14] use a SAT engine for identifying (or approximating) the minimal set of predicates needed to eliminate a set of spurious counterexamples during refinement of abstract C programs. The predicate minimizing algorithm is implemented in the MAGIC tool, which uses a theorem prover to compute predicate abstraction. The problem of finding small sets of predicates (yet not minimal) is also being investigated in the context of hardware designs in [15].

To our best knowledge, the technique reported in this paper is the first effort to apply lazy abstraction with an imperative predicates set for the actual construction of a predicate abstraction of software. The reported technique is defined in the context of MISRA-C programs.

Lazy Abstraction with Imperative Predicates

Counterexample Guided Abstraction Refinement. The CEGAR framework is shown in Fig 1:

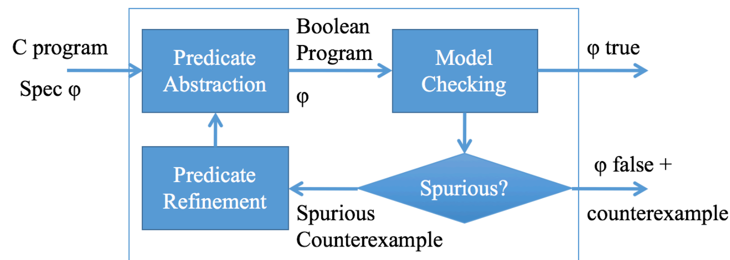

Fig. 1 Counterexample Guided Abstraction Refinement

Step 1. *Program abstraction.* Given a set of predicates, a finite state model is extracted from the code of a software system and the abstract transition system is constructed.
Step 2. *Verification.* A model checking algorithm is run in order to check if the model created by applying predicate abstraction satisfies the desired behavioral claim φ. If the claim holds, the model checker reports success (φ is *true*) and the CEGAR loop terminates. Otherwise, the model checker extracts a counterexample and the computation proceeds to the next step.
Step 3. *Counterexample validation.* The counterexample is examined to determine whether it is spurious. If this is the case, the bug is reported (φ is *false*) and the CEGAR loop terminates. Otherwise, the CEGAR loop proceeds to the next step.
Step 4. *Predicate refinement.* The set of predicates is changed in order to eliminate the detected spurious counterexample. Given the updated set of predicates, the CEGAR loop proceeds to Step 1.
Lazy Abstraction with Imperative Predicates. Lazy abstraction with interpolation-based refinement has been shown to be a powerful technique for verifying imperative programs. The lazy abstraction concept is aimed at optimizing the naïve abstract-check-refine loop by integrating the three steps. The lazy abstraction is based on the following two principles:

1 On-the-fly abstraction: The general approach generates the entire abstract model at the "Abstract" stage. However, some abstracted regions may never be visited (e.g. unreachable regions). The lazy abstraction concept suggests abstracting a region only when it is needed in the next step of checking. In this case, the abstraction task is driven by the checking process.

2 On-demand refinement: The lazy abstraction concept suggests that we can reuse the partial answer that is obtained in previous iterations. As a result, we can avoid refining those regions that have already been proved to be safe. Refinement is applied starting from the earliest state at which the abstract counterexample fails to have a concrete counterpart. This state is called pivot state.

Lazy Abstraction is based on a CEGAR loop. At the "Refinement" stage, the refinement is done by adding predicates that eliminate the new spurious counterexample while maintaining the predicates that were found in previous iterations. This is done by generating an interpolant for the path to the error state. Since each predicate corresponds to a Boolean state variable in the abstract model, the number of predicates directly determines the complexity of building and checking the abstract model. Let $|P|$ be the size of a given program and $|Pred|$ be the number of predicates in the abstraction refinement process. Computation cost of this model is $|P| \cdot 2^{|Pred|}$. It is obvious that the smaller the number of predicates in the abstraction refinement process, the exponential reduction in the cost of abstraction computation and model checking.

The abstraction refinement algorithm described in Algorithm 1 performs the lazy predicate abstraction analysis with a process to eliminate all the redundant predicates. First, the algorithm builds a CFA according to the MISRA-C program. Then it constructs a symbolic representation of the concrete transition relation by applying symbolic simulation techniques with the CFA. Next, we add predicates (initially an empty set) in current and next state form to the relation between variables, resulting in a Boolean formula. Finally, we enumerate symbolically on the values of the predicates, using a SAT solver. In fact, in order to prove the safety property, lazy abstraction constructs an *abstract reachability tree* (ART). Each node of the ART is labeled with a location of a CFA. Because the locations of a CFA will be visited at least once while constructing the ARG, the Boolean formula constructed in advance will be reused. When the abstract program needs to be refined, we use the same formula that we have already created, together with the new set of predicates, to create the new

Algorithm 1. lazy abstraction with imperative predicates

Input: program Π, safety property φ
Output: TRUE if proved $\Pi \vDash \varphi$, a counterexample if proved $\Pi \nvDash \varphi$, UNKNOWN otherwise
Declare:
T : set of spurious counterexamples
P : set of imperative predicates
L : size limit of P
Begin
build the CFA of program Π;
construct a symbolic representation of the concrete transition relation and Boolean formula between adjacent states in CFA;
while TRUE **do**
lazy predicate abstraction to prove the safety property φ;
if the result is TRUE
return TRUE
else
let τ be the abstract counterexample
if τ corresponds to a concrete counterexample
return FALSE
if $|P|$ exceed the predicate size limit L
return UNKNOWN**;**
$T := T \cup \{\tau\}$;
P := set of imperative predicates that eliminates all elements of T
update the Boolean formula between adjacent states with P
End

abstraction. The *while* loop is based on Counterexample-Guided Abstraction Refinement. If the lazy predicate analysis has exhaustively checked all program states and did not reach the error, then the algorithm terminates and reports that the program is safe. If the algorithm finds an error in the abstract state space, then the exploration algorithm stops and returns the counterexample. Now the according abstract error path is extracted from the counterexample and analyzed for feasibility. If the abstract error path is feasible, then this error path represents a violation of the specification and the algorithm terminates, reporting a bug. If the error path is infeasible, i.e., not corresponding to a concrete program path, then the precision was too coarse and needs to be refined.

The algorithm uses Craig interpolant to extract new predicates that must be added to the abstraction in order to rule out the infeasible error path. Instead of simply add the new predicates to P, the algorithm checks whether there are any unnecessary predicates that are useless to eliminate all the counterexamples and get rid of these redundant predicates.

Imperative Predicates Set. With the following problem: given a set of spurious counterexamples T and a set of candidate predicates P, find a minimal set $p \subset P$ which eliminates all the traces in T. [8] proposed a three step algorithm: first, find a mapping $T \mapsto 2^{2^{|P|}}$ between each trace in T and the set of sets of predicates in P that eliminate it. Second, encode each predicate $p_i \in P$ with a new Boolean variable p_i^b. Third, derive a Boolean formula σ, based on the predicate encoding, that represents all the possible combinations of predicates that eliminate the elements of T. For any satisfying assignment to σ, the predicates whose Boolean encodings are assigned $TRUE$ are sufficient for eliminating all elements of T. From the various possible satisfying assignments to σ, we look for the one with the smallest number of positive assignments. This assignment represents the minimal number of predicates that are sufficient for eliminating T.

The algorithm is intent on finding the minimal set $p \subset P$ and experiments show that predicate minimization can reduce both verification time and memory needed for verification. However, minimization does not mean that it is the best strategy. For example, some predicates that are eliminated before still can be added by the next CEGAR loop again. So, for each predicate p, let $enter(p)$ denote the number of time when p is added after being eliminated in the abstraction refinement loop. If $enter(p)$ exceeds a certain user-defined threshold *TH*, then p is assigned a dedicated state variable and can never be eliminated since then. If *TH* = 0, then every predicate will be assigned a dedicated state variable as soon as it is discovered. This is similar to performing abstraction with no predicate elimination. On the other hand, if *TH* is big enough, then the reentrant times of every predicate will be not under consideration. For any reasonable value of *TH,* we have a hybrid of predicate elimination with and without dedicated predicate states.

Refinements

Initial Predicates. Since in lazy abstraction, initial abstraction is typically coarse, the abstract search is very likely to reach the target (i.e. error states). Moreover, if the user guarantees that some error states will never be reached, the abstraction of these states is unnecessary. See Fig. 2 below, if the user promises that variable $LOCK$ will always be 0, then the left side sub-tree should be omitted instantly. The predicate $LOCK = 0$ can be added to the initial predicates set by user. Such a user provided initial predicate improves the efficiency significantly in the initial iterations.

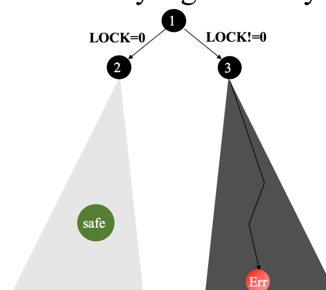

Fig. 2 Example of Guaranteed Safe States

Pointer Alias Analysis. In a typical application there may be a large number of variables having the correct type as $*p$, while only a few that p can actually point to. In order to minimize the size of the equation generated we use all the information we can extract from the program about the possible targets of p. Using the (dynamic) information obtained from the predicates, we can save a lot more than by merely using static points-to algorithms.
Let $\Theta(p)$ denote the set of variables which p can legally point to (i.e., the variables with a compatible type). We analyze the set of predicates P and extract a set of variables $\theta(p,P) \subseteq \Theta(p)$ such that $v \in (p,P)$ holds if it is possible to derive from the predicates that p points to v.
Let $P = \{\pi_1, \dots, \pi_k\}$ be the set of predicates. Then $\theta(p,P)$ is the set of variables $v \in (p,P)$ for which there exists a truth assignment to the predicates such that the resulting conjunction implies that p holds the address of v.

$$\theta(p,P) \triangleq \{v \in \Theta(p) \mid \exists b_1, \dots, b_k. \wedge_{i=1,\dots,k}(b_i \leftrightarrow \pi_1) \Rightarrow (p = \&v)\} \quad (1)$$

A pointer dereference $*p$ in an expression is replaced by a case split on all the variables from $\theta(p,P)$. Let $\theta(p,P) = \{v^1, \dots, v^k\}$. We replace every occurrence of $*p$ with

$$(p == \&v^1)?\, v^1 : (p == \&v^2)?\, v^2 : \dots (p == \&v^k)?\, v^k : \bot \quad (2)$$

where $\bot$ is a default value, which is never used. For example, assume we have predicate set $\{p = \&x, p = \&y\}$, the expression x = *p + 1 will be transformed to x = ((p == &x) ? x : y) + 1.

Experiments

All the experiments reported in this section have been carried out on a 32-bit Linux distribution (Ubuntu 14.04) running on an Intel Core i7-2760QM Processor (4 cores). We set the memory limit to 4 GB and the time limit to 3 hours. We use the benchmarks that are same with [14] in order to compare the performance of out tool with MAGIC and BLAST. Here we only compare BHLA(TH=2) with BLAST.

Table.1 Experiment Result for BLAST and BHLA

Program	BLAST				BHLA(TH=2)			
	Time	Iter	Pred	Mem	Time	Iter	Pred	Mem
funcall-nes	1	3	13/10	×	1	2	12/1	×
fun lock	2	7	7/7	×	2	4	7/3	×
driver.c	1	4	3/2	×	3	5	3/2	×
read.c	3	11	20/11	×	4	2	18/1	×
socket-y-01	2	13	16/6	×	3	3	16/2	×
opttest.c	4936	38	37/37	231	133	27	37/4	213
ssl-srvr-1	1523	16	33/8	175	145	15	32/2	138
ssl-srvr-2	452	13	2/1	60	123	15	2/1	58
ssl-srvr-3	785	14	32/7	103	98	12	32/2	128
ssl-srvr-4	160	11	27/5	44	84	10	27/2	38
ssl-srvr-5	1323	19	52/5	71	593	13	50/2	63
ssl-srvr-6	*	39	90/10	805	*	24	90/3	590
ssl-srvr-7	231	11	76/9	37	82	11	76/2	38
ssl-srvr-8	*	25	35/5	266	92	13	35/3	220
ssl-srvr-9	212	10	76/9	36	173	20	76/4	38
ssl-srvr-10	6785	20	35/8	148	111	14	35/3	138
ssl-srvr-11	368	11	78/11	51	224	24	78/3	57
ssl-srvr-12	1708	21	80/8	120	152	18	80/3	104
ssl-srvr-13	432	12	79/12	54	300	29	79/4	58
ssl-srvr-14	9548	27	84/10	278	206	20	83/3	252
ssl-srvr-15	*	31	38/5	436	103	11	36/3	401
ssl-srvr-16	*	33	87/10	480	219	19	84/3	358
ssl-clnt-1	258	16	5/3	43	88	13	5/2	31
ssl-clnt-2	389	15	28/4	52	101	18	28/2	35
ssl-clnt-3	310	14	5/4	49	124	21	4/2	39
ssl-clnt-4	273	13	4/3	45	119	19	4/2	29
TOTAL	29702	447	1178/221	3584	3458	382	1025/64	3026

Table. 1 show the experiment result for BLAST and BHLA respectively. ‘*’ indicates running time longer than 3 hours, ‘×’ indicate negligible values. Iter reports the number of iterations through the CEGAR loop. In Pred column, the first number is the total number of predicates discovered and the second is the final size of P.

From the experiment result we can see that both the execution time and memory usage of BHLA are less than BLAST. Moreover, there are three test cases BHLA can prove successfully while BLAST exceeds the time limit.

Summary

Predicate abstraction is a common and efficient technique in software verification domain. However, when the size of the program (number of branching conditions) is large, predicate abstraction suffers from the computation cost that increases exponentially as the number of predicates increases. Lazy abstraction with interpolation-based refinement has been shown to be a powerful technique for verifying imperative programs. In this paper, we propose a technique that combines lazy abstraction with imperative predicates set and realize it inside the BHLA tool. Experiment results show that our technique outperforms existing methods.

In the future, we would like to extend our abstraction refinement with more than one counterexample. Moreover, the priority-based search dedicated to improve the efficiency of the CEGAR iterations [15] is worth further investigation.

References

[1] Hennell Mike.MISRA C role in the bigger Picture of critical software development,2001.http://www.misrac2.com.

[2] Clarke E M, Grumberg O, Peled D. Model checking[M]. MIT press, 1999.

[3] Colón M A, Uribe T E. Generating finite-state abstractions of reactive systems using decision procedures[C]//Computer Aided Verification. Springer Berlin Heidelberg, 1998: 293-304.

[4] Graf S, Saïdi H. Construction of abstract state graphs with PVS[C]//Computer aided verification. Springer Berlin Heidelberg, 1997: 72-83.

[5] T, Rajamani S K. Boolean programs: A model and process for software analysis[R]. Technical Report 2000-14, Microsoft Research, 2000.

[6] E, Grumberg O, Jha S, et al. Counterexample-guided abstraction refinement[C]//Computer aided verification. Springer Berlin Heidelberg, 2000: 154-169.

[7] Armando A, Benerecetti M, Mantovani J. Abstraction refinement of linear programs with arrays[M]//Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 2007: 373-388.

[8] Cousot P. Abstract interpretation[J]. ACM Computing Surveys (CSUR), 1996, 28(2): 324-328.

[9] Kurshan R P. Computer-aided verification of coordinating processes: the automata-theoretic approach[M]. Princeton university press, 2014.

[10] Ball T, Rajamani S K. Boolean programs: A model and process for software analysis[R]. Technical Report 2000-14, Microsoft Research, 2000.

[11] Chaki S, Clarke E M, Groce A, et al. Modular verification of software components in C[J]. Software Engineering, IEEE Transactions on, 2004, 30(6): 388-402.

[12] Henzinger T A, Jhala R, Majumdar R, et al. Lazy abstraction[C]//ACM SIGPLAN Notices. ACM, 2002, 37(1): 58-70. Gulavani B S, Henzinger T A, Kannan Y, et al.

[13] SYNERGY: a new algorithm for property checking[C]//Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering. ACM, 2006: 117-127.

[14] Chaki S, Clarke E, Groce A, et al. Predicate abstraction with minimum predicates[M]//Correct Hardware Design and Verification Methods. Springer Berlin Heidelberg, 2003: 19-34.

[15] Clarke E, Grumberg O, Talupur M, et al. Making predicate abstraction efficient[C]//Computer Aided Verification. Springer Berlin Heidelberg, 2003: 126-140.