Goanna: Syntactic Software Model Checking

Ansgar Fehnker, Jörg Brauer, Ralf Huuck, and Sean Seefried

National ICT Australia Ltd. (NICTA)*
Locked Bag 6016
University of New South Wales
Sydney NSW 1466, Australia

Abstract. Goanna is an industrial-strength static analysis tool used in academia and industry alike to find bugs in C/C++ programs. Unlike existing approaches Goanna uses the off-the-shelf NuSMV model checker as its core analysis engine on a syntactic flow-sensitive program abstraction. The CTL-based model checking approach enables a high degree of flexibility in writing checks, scales to large number of checks, and can scale to large code bases. Moreover, the tool incorporates techniques from constraint solving, classical data flow analysis and a CEGAR inspired counterexample based path reduction. In this paper we describe Goanna's core technology, its features and the relevant techniques, as well as our experiences of using Goanna on large code bases such as the Firefox web browser.

1 Introduction

Model checking and static analysis are automated techniques promising to ensure (limited) correctness of software and to find certain classes of bugs automatically. One of the drawbacks of software model checkers [1–3] is that they typically operate on a low level semantic abstraction making them suitable for small code bases, but less so for larger software and, when soundness is paramount, are not applicable to industrial C/C++ code containing pointer arithmetic, unions, templates and alike. On the other hand, static analysis tools [4] have been concentrating on a shallower but more scalable and applicable analysis of large code bases. Typically, soundness is sacrificed for performance and practicality [5].

There are, however, many advantages in using a model checker. Specifications can often be given elegantly in temporal logic, there are many built-in optimizations in state-of-the-art tools, and especially CTL model checkers have been shown to be rather insensitive to the number of different checks performed on the same model.

In this work we present Goanna, a tool for static program analysis that makes use of the advances of modern model checkers and combines it with constraint

* National ICT Australia is funded by the Australian Governments Department of Communications, Information Technology and the Arts and the Australian Research Council through Backing Australias Ability and the ICT Research Centre of Excellence programs.

solving and counterexample based abstraction refinement (CEGAR) techniques [15, 1]. Goanna uses standard symbolic CTL model checking as implemented in the NuSMV [6] tool on a high-level program abstraction. This abstraction includes the control flow graph (CFG) of a program and labels (atomic propositions) consisting of syntactic occurrences of interest. On this level of abstraction model checking is fast, scalable to large code fragments and scalable to many such checks in the same model. Given that there are typically only a few bugs in every thousand lines of code [7] the abstraction is also appropriate for a first approximation. In a second step, more advanced features are used such a constraint solving and CEGAR-inspired path reduction to exclude false alarms. On top we incorporated alias analysis and summary-based interprocedural analysis to gain additional depth while remaining scalable.

In Section 2 we briefly explain the underlying technology, followed by a list of additional features in Section 3. A summary of our experiences from analyzing industrial code can be found in Section 4.

2 Core Technology

Goanna is built on an automata based static analysis framework as described in [8], which is related to [9–11]. The basic idea of this approach is to map a C/C++ program to its CFG, and to label this CFG with occurrences of syntactic constructs of interest automatically. The CFG together with the labels can be seen as a transition systems with atomic propositions, which can easily be mapped to the input language of a model checker, in our case NuSMV, or directly translated into a Kripke structure for model checking.

A simple example of this approach is shown in Fig. 1. Consider the contrived program `foo` which is allocating some memory, copying it a number of times to `a`, and freeing the memory in the last loop iteration.

One example of a property to check is that after freeing some resource it will not be used, i.e., otherwise indicating some memory corruption. In our approach we syntactically identify program locations that allocate, use, and free resource p. This is done automatically by pattern matching for the pre-defined relevant constructs on the program's abstract syntax tree. Next, we automatically label the program's CFG with this information as shown on the right hand side of Fig. 1 and encode the check itself as follows in CTL:

$$AG\ (malloc_p \Rightarrow AG\ (free_p \Rightarrow \neg EF\ used_p)),$$

which means that whenever there is `free` after `malloc` for a resource p, there is no path such that p is used later on. Neglecting any further semantic information will lead to a false alarm in the current example since p is only freed once in the last loop iteration and there is no access to it later. However, the abstraction in Fig. 1 does not reflect this. We will come back to this issue in Section 3.2.

One of the advantages of the proposed approach is that, e.g., a stronger variant of the above check can easily be obtained by switching from EF to AF, i.e., warning only when something goes wrong on all paths. As such, CTL model

```
void foo() {
   int x, *a;
   int *p=malloc(sizeof(int));
   for(x = 10; x > 0; x--) {
     a = *p;
     if(x == 1)
       free(p);
   }
 }
```

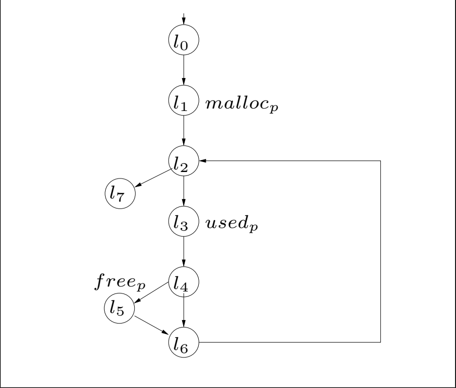

Fig. 1. Example program and labeled CFG for use-after-free check.

checking is an easy and powerful approach to defining different properties with different strength quickly.

3 Features

The aforementioned static analysis approach allows for describing properties in a simple straightforward fashion. In this section we present a number of more advanced features integrated in Goanna. In particular, we describe new techniques to increase the precision of the analysis and the type of bugs that can be found by the tool.

3.1 Constraint Analysis

The model-checking based static analysis approach described in Section 2 is well suited for checking control-flow dependent properties such as certain types of memory leaks, uninitialized variables, potential null-pointer dereferences or alike. However, one of the major concerns for software correctness as well as software security are buffer overruns such as accesses to arrays beyond their bounds.

Typically, these problems have been addressed by abstract interpretation solutions [12]. In Goanna we added an interval constraint solving approach that uses the approach first developed by [13]. These techniques guarantee a precise least solution for interval analysis including operations such as additions, subtraction and multiplication without any widening in the presence of (simple) loops. Goanna uses interval analysis to infer for every integer variable in every program statement its potential range and uses it to, e.g., check for buffer over- and underruns.

3.2 False Path Reduction

Not every bug is a real one, i.e., the initial coarse syntactic abstraction might lead to a number of false positives. For instance in the example in Figure 1 the memory will only be accessed as long as the memory is not freed. A straightforward flow-sensitive analysis will not catch this fact.

In Goanna we make use of the aforementioned interval constraint solving approach to rule out some of those spurious alarms automatically [14]. For a given counterexample path Goanna subjects it to the interval analysis as described in the previous Section 3.1. This means, for every variable every possible value will be approximated along the counterexample path. E.g., in the program of Figure 1 the counterexamples loops trough the program after condition `x==1` becomes true. However, when $x = 1$, the loop counter will be set to $x = 0$ in the next iteration, invalidating the loop condition of $x > 0$ and preventing the loop body to be re-entered. An interval constraint analysis over the counterexample path will come to the same conclusion by discovering that here is no possible satisfying valuation for x.

Goanna will learn this fact and create an observer automaton as described in [14]. This observer rules out a minimum set of conflicts, i.e., those conditions responsible for a false path. Observers are put in parallel composition to the existing coarse grained model to obtain a more precise model showing fewer false positives. While the parallel composition adds some overhead to the overall computation time, it is only added in a few cases were spurious alarms are found, ensuring an overall efficient analysis.

Similar to CEGAR the above process is iterated until no more bugs are found or no more counterexample paths can be eliminated. Note that since the interval analysis is an approximation itself, it is not guaranteed that all false positives can be removed.

3.3 Interprocedural Analysis

The aforementioned analysis techniques are mostly applicable for intraprocedural analysis, i.e., analyzing one function at a time. While this is sufficient for returning good results on standard software, it neglects, e.g., null pointers being passed on through several functions and then finally dereferenced without being properly checked. Therefore, we developed a summary-based interprocedural analysis. This analysis includes two features: The computation of alias information by inferring points-to sets [16] and computing a summary based on the lattice of the property under investigation. In the case of passing null pointer information, the summary records for every passed pointer if it points to Null, to something not Null or to an unknown value.

Given the call graph of the whole program Goanna computes the fixed point on the summary information, rerunning the local analysis when needed. Even in the presence of recursion this procedure typically terminates within two to three iterations involving a limited set of functions.

3.4 Additional Features

We briefly summarize a number of additional features built into Goanna, which are mostly on a usability level.

Integration. Goanna can be run from the command line as well as be tightly integrated in the Eclipse IDE. In the latter case, all the warnings are displayed in the IDE, settings and properties can be set in the IDE and for every bug a counterexample trace created by the model checker can be displayed. As a consequence, Goanna can be used during software development for every compilation.

Incremental analysis. To minimize the analysis overhead Goanna creates hashes of every function and object. If those functions and objects are not changed between compilations there will be no re-analysis. Of course, hashes are insensitive to additional comments, line breaks and minor code rearrangements.

User defined checks. Goanna comes with a simple language including CTL-style patterns, which enables the user to define his own checks. The language builds on pre-defined labels for constructs of interest. This language does not enable full flexibility, but it is safe to use and covers most scenarios. A full abstract language is in preparation.

4 Experiences

We have been evaluating Goanna on numerous open source projects as well as on industrial code. The largest code base has been the Firefox web browser, which has 2.5 million lines of code after preprocessing. Run-time is roughly 3 to 4 times slower than compilation itself for intraprocedural analysis. Interprocedural analysis can double the run-time, but it is worth to mention, that most of the time is spent in the alias analysis that is not yet efficiently implemented.

In a typical analysis run over 90% of files are analyzed in less than 2 seconds. Roughly 40% of the analysis time is spent for model checking, 30% for interval analysis and 30% for pattern matching, parsing and other computations. Adding more checks gives modest penalties, i.e., a threefold increase in the number of checks doubles the analysis time. Interval analysis is fast as long as the number of constraints is below a few hundred, i.e., resulting in a maximal overhead of one second.

In very rare cases, due to C/C++ macros or C++ templates a function might contain several hundreds of variables and very long code fragments. In these cases the overall analysis might take considerable time. However, in our experience from analyzing the source code of Firefox a time out of 2 minutes was only exceeded once out of roughly $250,000$ functions for the complete analysis.

The overall defect density is between 0.3 to 2 bugs per 1000 lines of code, depending on the code base and type of checks enabled. This is comparable with other commercial static code checkers [4].

References

1. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Proc. TACAS 2005. LNCS 3440 (2005) 570–574
2. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proc. TACAS 2004. LNCS 2988 (2004) 168–176
3. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL. (2004) 232–244
4. Emanuelsson, P., Nilsson, U.: A comparative study of industrial static analysis tools. Electronic notes in theoretical computer science (2008)
5. Engler, D., Chelf, B., Chou, A., Hallem, S.: Checking system rules using system-specific, programmer-written compiler extensions. In: Proc. Symposium on Operating Systems Design and Implementation, San Diego, CA. (October 2000)
6. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In: Intl. Conf. on Computer-Aided Verification (CAV 2002). Volume 2404 of LNCS., Springer (2002)
7. scan.coverity.com: Open source report. Technical report, Coverity Inc. (2008)
8. Fehnker, A., Huuck, R., Jayet, P., Lussenburg, M., Rauch, F.: Model checking software at compile time. In: Proc. TASE 2007, IEEE Computer Society (2007)
9. Holzmann, G.: Static source code checking for user-defined properties. In: Proc. IDPT 2002, Pasadena, CA, USA (June 2002)
10. Dams, D., Namjoshi, K.: Orion: High-precision methods for static error analysis of C and C++ programs. Bell Labs Tech. Mem. ITD-04-45263Z, Lucent Technologies (2004)
11. Schmidt, D.A., Steffen, B.: Program analysis as model checking of abstract interpretations. In: Proc. SAS '98, Springer-Verlag (1998) 351–380
12. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas, ACM Press, New York, NY (1979) 269–282
13. Gawlitza, T., Seidl, H.: Precise fixpoint computation through strategy iteration. In: ESOP. LNCS 4421, Springer Verlag (2007) 300–315
14. Fehnker, A., Huuck, R., Seefried, S.: Counterexample guided path reduction for static program analysis. In: Correctness, Concurrency, and Compositionality. Volume number to be assigned of Festschrift Series., LNCS (2008)
15. Henzinger, T., Jhala, R., Majumdar, R., SUTRE, G.: Software verification with BLAST. In: Proc. SPIN2003. LNCS 2648 (2003) 235–239
16. Andersen, L.: Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, Unversity of Copenhagen (1994)