Submitted to:11th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2006)

Goanna — A Static Model Checker

Ansgar Fehnker[1], Ralf Huuck[1], Patrick Jayet[2⋆], Michel Lussenburg[2⋆], and Felix Rauch[1]

[1] National ICT Australia Ltd. (NICTA)[⋆⋆] and University of New South Wales,
Locked Bag 6016, NSW 1466, Australia

[2] Department of Computer Science, Swiss Federal Institute of Technology (ETH),
CH-8092 Zurich, Switzerland

Document Revision: 1.11 Date: 2006-06-05 04:30:37

Abstract. In this work we present Goanna, the first tool that uses an off-the-shelf model checker for the static analysis of C/C++ source code. We outline its architecture and show how syntactic properties can be expressed in CTL. Once the properties have been defined the tool analyses source code automatically and efficiently. We demonstrate its applicability by presenting experimental results on analysing OpenSSL and the GNU coreutils.

1 Introduction

Formal design and analysis techniques are successfully applied to hardware. In fact, model checking parts of the chip design is common practice. In contrast, the application of industrial strength software design and verification technology has been much less successful. A lot of work has been done in the area of model driven design. In particular, the synchronous community has delivered powerful tools for the specification and subsequent (limited) verification of software. Also, semi-formal description techniques such as UML are widely used. However, the application of verification technology to existing and complex software has been much less successful.

The reasons are manifold: Full formal verification as done by interactive theorem proving is expensive. It requires a lot of time and expertise, making it often impractical for software that has a short life cycle, is not highly safety-critical, or suffers from a high pressure to market. Algorithmic verification techniques have to deal with software's infinite state space, requiring abstraction techniques to make properties of interest decidable. Suitable abstractions are typically hard to

[⋆] This work was carried out while visiting NICTA.

[⋆⋆] 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.

compute and the overall interaction required by the user in order to apply them to real-life software are often considerable.

One area that has been successful is static analysis. Approaches such as abstract interpretation, data flow analysis and other static checking techniques have made it into several industrial strength tools.

In this work we present Goanna, a static analysis tool for C/C++ source code based on model checking. It uses the NuSMV model checker as the underlying verification engine, allows the specification of user defined properties and scales well to commercial size software. It does not require any user interaction making it particularly suited to be integrated into the software development process. Moreover, it is the first step of bringing static analysis and software model checking closer together by providing one uniform framework.

2 Technology

The basic ideas of solving static analysis problems by model checking have been first developed by Steffen and Schmidt [1]. While their main focus has been on developing a safe approximation of the program's behaviour, we have a stronger focus on the effectiveness of the analysis and trade this for soundness. As a result we use full CTL for checking syntactic program properties.

First we define atomic propositions of interest, e.g., whether a variable is declared, used, or assigned a value. For a variable named x we write $decl_x$, $used_x$ and $assigned_x$ for the respective propositions. We use a pattern matching approach to relate certain patterns on a program's abstract syntax tree (AST) with propositions of interest. In a second step we automatically extract the control flow graph (CFG) of a program and label it with the previously determined propositions.

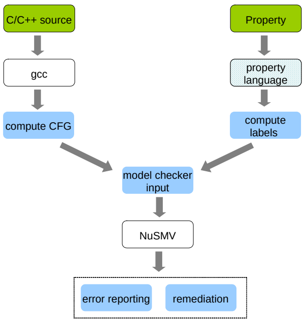
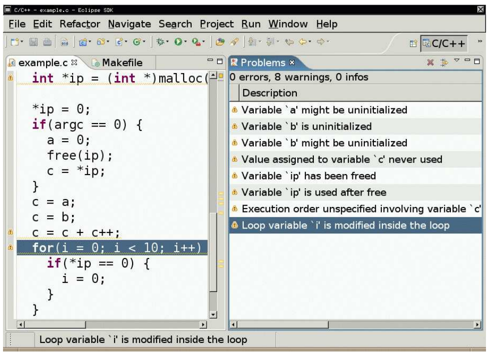


Fig. 1. Goanna architecture


C/C++ - example.c - Eclipse SDK
File Edit Refactor Navigate Search Project Run Window Help
C/C++
example.c
Makefile
Problems

```
int *ip = (int *)malloc(

*ip = 0;
if(argc == 0) {
  a = 0;
  free(ip);
  c = *ip;
}
c = a;
c = b;
c = c + c++;
for(i = 0; i < 10; i++)
  if(*ip == 0) {
    i = 0;
  }
}
```

0 errors, 8 warnings, 0 infos

Description
Variable `a' might be uninitialized
Variable `b' is uninitialized
Variable `b' might be uninitialized
Value assigned to variable `c' never used
Variable `ip' has been freed
Variable `ip' is used after free
Execution order unspecified involving variable `c'
Loop variable `i' is modified inside the loop

Loop variable `i' is modified inside the loop

Fig. 2. Eclipse Embedding

The translation of an annotated CFG into an NuSMV model is rather straightforward and the encoding can be done in an efficient way resulting in a small state space. Properties of interest can then be expressed as CTL formulae over this model. E.g., checking for uninitialised variables can be expressed as follows:

$$\text{AG } \mathit{decl}_x \Rightarrow (\text{A } \neg \mathit{used}_x \text{ W } \mathit{assigned}_x)$$

This means we require that on all program paths if a variable is declared it must not be used until it has a value assigned or it will not be used at all. We use the weak until operator W here to include the second possibility. The latter can also point to unused variables, which is checked separately.

Our tool chain is depicted in Figure 1. We use gcc as a front end, as one of its features allows us to easily output the AST of C/C++ programs in an intermediate language. We parse the AST and on the one hand generate the CFG from it and on the other hand match patterns on the AST, which constitute the atomic propositions of a CTL formula expressing the desired property. We label the CFG with atomic propositions where their respective patterns where matched. Once the patterns and the CTL formula have been specified, the translation of the C/C++ source code into a suitable NuSMV model and its checking is fully automatic.

The current implementation is developed in OCaml. It integrates in Makefiles and thus automatically supports development environments such as Eclipse. A screen shot of Goanna running in combination with Eclipse can be found in Figure 2. This enables a seamless integration into the overall software development process.

3 Application

To evaluate the applicability of our tool, we examine two real-world open-source software packages: The GNU coreutils[3], which provide basic file, shell and text manipulation utilities (59 kLoC[4]), and the *OpenSSL*[5] toolkit implementing the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols (260 kLoC). We analyse the source code of these two packages on a DELL PowerEdge SC1425 server, with a 3.4 GHz Xeon processor and 1.5 GiB of memory.

Analysing the whole source with our current Goanna tool (which has not yet been optimised) took slightly less than 2 minutes for the coreutils and slightly less than 29 minutes for OpenSSL. The latter is somewhat distorted by a single pathological file that takes almost 12 minutes to analyse. In practice, analysis times are typically much shorter, not only because the analysis can be done incrementally on the set of recently changed files only, but also because a more in-depth study of Goanna's analysis times shows that a large majority of source file is analysed quite quickly. In fact, 72% of all source files in the coreutils are

[3] http://www.gnu.org/software/coreutils/

[4] LoC = Lines of Code, kLoC = 1000 Lines of Codes

[5] http://www.openssl.org/

analysed in less than 1 second and 95% under 5 seconds. Similarly, for OpenSSL 83% of all files are analysed in under 1 second and again 95% under 5 seconds.

Note that the current prototype has not yet been optimised for lower execution times. There is still a lot of room for optimisations, for example by optimising the way we search the AST for interesting patterns (XXX), the library we use to conduct the search on the AST (which is convenient but not fast), or by reducing the chunk size of the source code that NuSMV is invoked with.

Looking at the memory requirements of our tool we find that the maximum memory consumption of the analysis is about 65 MiB to analyse the coreutils and about 113 MiB for OpenSSL respectively. This is in both cases much below the limit set by todays PCs used by developers.

The above numbers show that the tool is already quite usable in practice. A full evaluation of course requires also an analysis of the precision of the tool, with looking at the number of real bugs found and the number of false positives. Such a study is very time consuming and we are still in the process of qualitatively evaluating Goanna regarding its precision. Preliminary results indicate that the precision of our approach is comparable to standard static analysis.

4 Conclusion

In this work we presented Goanna, the first static analyser purely based on an off-the-shelf model checker. We demonstrated that the approach scales well to real-life software making it suitable for the integration into the overall software development process.

While Goanna is fast, it is not yet more precise than traditional static analysis. However, we anticipate to improve on this by incorporating more semantic-based software model checking techniques such as predicate abstraction. The foundation of this integration has been laid by having a uniform framework for static analysis as well as traditional model checking.

References

1. Schmidt, D.A., Steffen, B.: Program analysis as model checking of abstract interpretations. In: SAS '98: Proceedings of the 5th International Symposium on Static Analysis, London, UK, Springer-Verlag (1998) 351–380