Goanna – Fast, Flexible Static Analysis with OCaml

Mark Bradley, Franck Cassez, Ansgar Fehnker,
Thomas Given-Wilson and Ralf Huuck

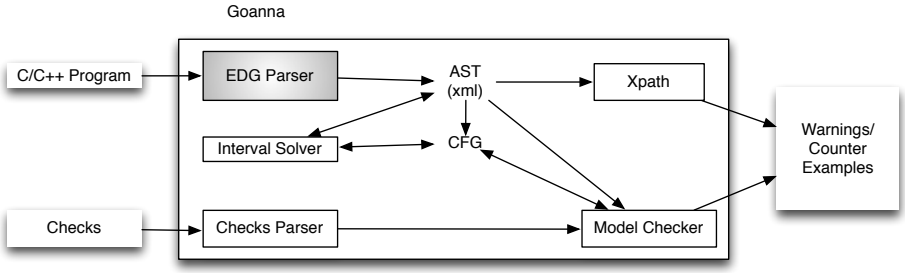
Abstract

We present our experiences with developing, improving, and maintaining an industrial strength C/C++ static analysis tool whose core is written in OCaml. In particular we focus on integrating several different components with various backgrounds and interfaces, as well as our own domain specific languages.

Goanna Overview

Goanna is a static analysis tool for detecting programming flaws and security vulnerabilities in C/C++ source code. The core of Goanna uses OCaml to combine several diverse components and efficiently manage their interaction. This includes: parsing with the EDG commercial C/C++ parser, managing and annotating the abstract syntax tree (AST) in XML with XPath, building a custom control flow graph (CFG), model checking over the combined AST and CFG, determining value intervals and ranges, and reachability constraints for variables. Handling all these (and other) components in an efficient and effective manner requires a combination of functional and stateful algorithms and optimisations.

Particular challenges come from the need to handle all these components and interactions in a manner that scales to around 150 different checks and industrial projects containing millions of lines of code.



Significant Components

Goanna has several components that come from independent origin and need to be handled together. The EDG parser is written in C and interfaces to Goanna through a custom interface written in OCaml. The AST is stored as XML which allows for XPath to easily discover certain patterns, and for extra information to be added in the form of custom XML tags and attributes.

Within Goanna's OCaml code the CFG is stored as custom data structures that allow for both forward and reverse lookups of the graph. In particular this supports a variety of properties that need to be checked (see next section for more). Similarly, integer, pointer, and boolean values can be handled by intervals (with a lower and upper bound, or infinity if unknown). This allows quick approximations of values within the program using an interval solver.

Challenges and Solutions

Integrating all the components of Goanna and achieving the necessary performance for industrial applications poses some challenges. Although OCaml proves effective for Goanna, some solutions are less than elegant.

Combining components, including across different languages (e.g. OCaml, C & C#) into a single application can be complex. Since `ocamlbuild` does not have sufficient flexibility, our build process involves many hand crafted scripts and dependencies on different compilers.

OCaml pattern matching is efficient for some parts of Goanna, particularly customised data structures. However, putting all the associated information into a single structure leads to overhead issues. Thus, we use several separate but associated structures and cross-reference to achieve best performance.

Similarly, handling large complex structures, such as the AST, can be very slow to query in a purely functional manner. Significant performance is gained by introducing stateful caching mechanisms into Goanna.

Many parts of the Goanna code exploit typical functional operations such as fold and map. Unfortunately OCaml's lack of fusion over these operations often limits performance. We were able to improve performance significantly by rewriting the Goanna code to compose functions before using them in, say a map or fold, over a list or set.

When compiling Goanna for several environments we exploited `camlp4` macros to customise the source code. However, the typing of these macros often required defining values when the macros weren't used, such as an empty string or unit value in a never used `ELSE` block.

Conclusions

OCaml proves to be an effective language for writing high performance industrial strength applications. Indeed, pattern matching, type safety, functional features, and comparative comprehensibility all improve the coding experience. However, some limitations in the build and compilation process can be improved to yield even better results in future.