A Language for Describing Formal Analyses

Cornel C. Izbaşa

Department of Computer Science, West University of Timişoara,
Institute e-Austria Timişoara,
`cornel@www.ieat.ro`

Abstract

Studying various projects in the domain of formal verification we encounter concepts like predicates, states, transitions, regions, abstraction, control-flow automata's, value-flow graphs, binary decision diagrams - concepts that are usually difficult to model in any programming language of choice. As a consequence, understanding and improving / extending an existing formal verification tool is often almost as hard as developing one. In my humble opinion, a possible solution to this problem would be the development of a programming language designed specifically for easing the development of formal verification tools.

Motivation

As I pointed out in the abstract, it can be hard to improve or extend formal verification tools because the concepts behind them aren't very easy to model. So, a possible approach would be to create a new programming language (independent or embedded in another language) for just this end: writing formal verification tools in it.

Existing formal verification tools like Berkeley's **BLAST** [`http://www-cad.eecs.berkeley.edu/~tah/blast/`] should correspond to programs in this new language that are a couple of hundred lines long. New ideas, possible improvements to the currently used verification algorithms should be much more easily implemented and tested for validity, as they would correspond to changing say 30-50 lines of code.

Having a specialized programming language leaves also room for multiple-strategy approaches. We could decide based on various criteria (maybe based on the software metrics tools of Mr. Radu Marinescu) which verification algorithm to employ for a specific segment / slice of a program. So we would have a programming language that's also general purpose, but with a great ability of modeling and manipulating *predicates*, *states*, *regions*, *transitions*, *abstraction* - everything useful for *labeled transition systems* and *control-flow automata*'s.

This type of approach would be also useful with projects like Stanford's **MetaL** [`http://www.stanford.edu/~engler/`] - we could devise a translator for our programs to **MetaL** checkers and viceversa once we have their source - or Reliable Software Technologies' **ITS** [`http://www.cigital.com/its4/`], **Splint** [`http://lclint.cs.virginia.edu/`] that rely on the conformance to *patterns* to give hints about the correctness of a particular piece of software and should certainly not be limited to modeling the concepts behind **BLAST**, even if for starters and as proof of feasibility we would like to see *lazy abstraction* implemented in a reasonably short program in our language. Ideally, we would have the functionality of as many verification algorithms as possible offered as libraries written in this language.

As a final reason for creating this language we note that approaches like this have also been successful in the realm of theorem provers which is related to the realm of formal verification.

Implementation ideas

The target language we chose for starters is C, due to it's extended usage in driver and operating system code, but more languages like Java and C++ should be testable as well after writing the appropriate filters that convert the sources into the corresponding *CFA*'s.

Given that an easy implementation for *lazy abstraction* is a sort of test objective for the validity of the idea, and taking into account all the advantages of functional programming languages, we chose **ML** for the implementation of our programming language. Specifically, Inria's **OCaml** [`http://caml.inria.fr`] which has the advantages that it's the implementation language of **BLAST** and of **CIL** [`http://manju.cs.berkeley.edu/cil/index.html`] - the simplified C library that **BLAST** uses and that we plan to use as well. **OCaml** has excellent lexical analyzer and parser creation facilities both internal and external (**ocamllex, ocamlyacc**). We will probably draw inspiration from **BLAST**'s **CIL**2*CFA* conversion mechanism - but this should go in an auxiliary library that comes with our language for easing the use of C code as target for verification and analysis.

We are also thinking about another language which would be the 'real' target language - a sort of textual representation of a *CFA*(CFA-TR). We would extend our target languages collection by writing 'filters' that convert the test sources into CFA-TR which we can use in our verification programs.

The end (or, hopefully, not so)

We hope to achieve a convergence with the **BLAST** project by implementing and keeping their tool up to date in our language over a shorter period of time and a convergence (merger?) with the **MetaL** project over a longer period of time, and quite possibly a convergence with Mr. Radu Marinescu's software metrics projects.