Abstract

Many current verification tools use a logical reasoning engine as a back-end, mostly SMT-solvers. While SMT-solvers are efficient and relatively predictable for quantifier free problems, they are not, in general, complete for problems with quantifiers, and tend to show unpredictable performance in the presence of quantifiers, often requiring expert attention for verification to succeed. More advanced software verification goals require the use of quantifiers in order to model concepts for which no decision procedure is implemented in the solver, such as modeling ownership or permissions on the heap.

Automated theorem provers (ATPs) based on the superposition calculus perform significantly better when reasoning with quantifiers, but are weaker for problems with large propositional parts - which many program verification problems contain - and for problems that include significant theory reasoning, such as linear integer arithmetic. A major difference between ATPs and SMT-solvers is that ATPs perform small steps that guarantee progress (generating new clauses), while SMT-solvers must find a conflict (a contradiction) in order to progress, and the number of steps to reach a contradiction is not bounded for problems with quantifiers - hence progress cannot be guaranteed for any number of steps.

In this thesis we develop a verification technique that embeds theorem proving deeply in the verifier. Our technique maintains a superposition-based theorem prover at each program point, and allows these provers to communicate relevant information in order to build a proof for the whole program. Our technique takes advantage of some properties of programs, such as the control flow structure and lexical scope, to restrict the proof search space. Each prover works incrementally in logical fragments of increasing strength and predictable worst-case performance. We define a hierarchy of logical fragments based on size measures of clauses and the derivation depth of clauses in the calculus. In our technique, each prover handles a much smaller problem than the verification of the entire program, and much of the control flow is handled directly, reducing the propositional part of the problem significantly.

We prove that our technique is complete for FOL without theories, and a weaker completeness result when lexical scoping is enforced. We have implemented our technique and, while we cannot prove many programs because of the lack of arithmetic support, our implementation can be used as a pre-processing step before an SMT-solver to reduce the overall proving time. Our implementation applies successively stronger logical fragments, each with predictable worst case run-time, and intermediate results (such as proven assertions) can be extracted at any time.