Cogent: Accurate theorem proving for program verification

Byron Cook — Microsoft Research
Daniel Kroening — ETH Zurich
Natasha Sharygina — Carnegie Mellon University

Abstract. Many symbolic software verification engines such as Slam and ESC/Java rely on automatic theorem provers. The existing theorem provers, such as Simplify, lack precise support for important programming language constructs such as pointers, structures and unions. This paper describes a theorem prover, Cogent, that accurately supports all ANSI-C expressions. The prover's implementation is based on a machine-level interpretation of expressions into propositional logic, and supports finite machine-level variables, bit operations, structures, unions, references, pointers and pointer arithmetic. When used by Slam during the model checking of over 300 benchmarks, Cogent's improved accuracy reduced the number of Slam timeouts by half, increased the number of true errors found, and decreased the number of false errors.

1 Introduction

Program verification engines, such as symbolic model checkers and advanced static checking tools, often employ automatic theorem provers for symbolic reasoning. For example, the static checkers ESC/Java [1] and Boogie [2] use the Simplify [3] theorem prover to verify user-supplied invariants. The Slam [4] software model-checker uses Zapato [5] for symbolic simulation of C programs. The Blast [6] and Magic [7] tools use Simplify.

Most automatic theorem provers used in program verification are based on either Nelson-Oppen or Shostak's combination methods. These methods combine various decision procedures to provide a rich logic for mathematical reasoning. However, when applied to software verification, the fit between the program verifier and the theorem prover is not ideal. The problem is that the theorem provers are typically geared towards efficiency in the mathematical theories, such as linear arithmetic over the integers. In reality, program verifiers rarely need reasoning for unbounded integers, and the restriction to linear arithmetic is too limiting. Moreover, because linear arithmetic over the integers is not a convex theory (a restriction imposed by Nelson-Oppen and Shostak), the real numbers are often used instead. Software programs, however, use the reals even less than they do the integers.

The program verifiers must provide support for language features that are not easily mapped into the logics supported by the existing theorem provers. These features include pointers, pointer arithmetic, structures, unions, and the potential relationship between these features. When using provers such as Simplify, the program verification tools typically approximate the semantics of these features with axioms over the symbols used during the encoding. However, using such axioms has a drawback—axioms can interact badly with the performance heuristics that are often used by provers during axiom-instantiation. Additionally, because bit vectors and arrays are not convex theories, many provers do not support them. In those that can, the link between the non-convex decision procedures can be unsatisfactory. As an example, checking equality between a bit-vector and an integer variable is typically not supported.

Another problem that occurs when using prover such as SIMPLIFY or ZAPATO is that, when a query is not valid, the provers do not supply concrete counterexamples. Some provers provide partial information on counterexamples. However, in program verification this information rarely leads to concrete valuations to the variables in a program, which is what a programmer most wants when a program verification tool reports a potential bug in the source code. For this reason, model checkers such as SLAM, BLAST, and MAGIC do not provide valuations to the program variables when an error trace is presented to the user.

This paper addresses the following question: *When verifying programs, can we abandon the Nelson-Oppen/Shostak combination framework in favor of a prover that performs a basic and precise translation of program expressions into propositional logic?*

We present a tool, called COGENT, that implements an eager and accurate translation of ANSI-C expressions (including features such as bitvectors, structures, unions, pointers and pointer arithmetic) into propositional logic. COGENT then uses a propositional logic SAT-solver to prove or disprove the query. Because COGENT's method is based on this eager translation to propositional logic, the models found by SAT-solvers can be directly converted to counterexamples to the original C input query. We evaluated COGENT's performance in SLAM, COMFORT [8], and BOOGIE. The experimental evidence indicates that COGENT's approach can be successfully used in lieu of conventional theorem provers.

2 Encoding into Propositional Logic

COGENT operates by eagerly translating expressions into propositional logic, and then using a propositional logic SAT-solver. COGENT is inspired by the success of CBMC and UCLID. UCLID encodes separation logic and uninterpreted functions eagerly into propositional logic. It does not, however, support bitvector logic. CBMC is a bounded model checker for C programming language and eagerly compiles bitvector arithmetic into propositional logic. COGENT is also used as a decision procedure for the SATABS [9] model checker. COGENT is a theorem prover intended for use in an abstraction framework such as SLAM or MAGIC, and thus, does not implement any abstraction by itself.

In hardware verification, the encoding of arithmetic operators such as shifting, addition, and even multiplication into propositional logic using arithmetic circuit descriptions is a standard technique. We use a similar approach in COGENT, with several modifications:

- In addition to the features supported by the existing tools, COGENT's translation allows the program verification tools to accurately reason about arithmetic overflow, bit operations, structures, unions, pointers and pointer arithmetic.

Model checking result	SLAM/ZAPATO	SLAM/COGENT
Property passes	243	264
Time threshold exceeded	39	17
Property violations found	17	19
Cases of abstraction-refinement failure	9	8

Fig. 1. Comparison of SLAM/ZAPATO to SLAM/COGENT on 308 device driver correctness model checking benchmarks. The time threshold was set to 1200 seconds.

- COGENT uses non-determinism to accurately model the ambiguity in the ANSI-C standard, i.e., for the representation of signed types. This differs from the support for bitvectors from theorem provers such as CVC-LITE [10].
- We use non-determinism to improve the encodings of some functions, such as multiplication and division, in a way that is optimized for SAT-solvers.

The technical details of COGENT's encoding for ANSI-C expressions including the use of non-determinism for accuracy and efficiency, can be found in [11].

3 Experimental evaluation

Experiments with symbolic software model checking. We have integrated COGENT with SLAM and compared the results to SLAM using its current theorem prover, ZAPATO. We ran SLAM/COGENT on 308 model checking benchmarks. The results are summarized in Fig. 1.

SLAM/COGENT outperforms SLAM/ZAPATO. Notably, the number of cases where SLAM exceeded the 1200s time threshold was reduced by half. As a result, two additional device driver bugs were found. The cases where SLAM failed to refine the abstraction [12] were effectively unchanged. During SLAM's execution, the provers actually returned different results in some cases. This is expected, as the provers support different logics. For this reason, there are queries that ZAPATO can prove valid and COGENT reports as invalid (e.g., when overflow is ignored by ZAPATO), and vice-versa (e.g., when validity is dependent on pointer arithmetic or non-linear uses of multiplication). Overall, we found that COGENT is more than 2x slower than ZAPATO. On 2000 theorem proving queries ZAPATO executed for 208s, whereas COGENT ran for 522s. Therefore, the performance improvement in Fig. 1 is indicative that, while COGENT is slower, COGENT's increased accuracy allows SLAM to do less work overall.

During the formalization of the kernel API usage properties that SLAM is used to verify [4], a large set of properties were removed or not actively pursued due to inaccuracies in SLAM's theorem prover. For this reason the results in Fig. 1 are not fully representative of the improvement in accuracy that SLAM/COGENT can give. In order to further demonstrate this improved accuracy, we developed and checked several new safety properties that could not be accurately checked with SLAM/ZAPATO. For more information on this property and a previously unknown bug that was found see [11].

Experiments with extended static checking We have also integrated COGENT with BOOGIE [2], which is an implementation of Detlef *et al.*'s notion of *extended static checking* [1] for the C# programming language. BOOGIE computes *verification conditions* that are checked by an automatic theorem prover.

We have applied COGENT to these verification conditions generated by BOOGIE and compared the performance to SIMPLIFY. The results were effectively the same. For more information on this application and, in particular, how we handle the nested quantifiers that appear in the BOOGIE queries, see [11]. We make [11], our tool, and bitvector benchmark files available on the web[1] in order to allow other researchers to reproduce our results.

4 Conclusion

Automatic theorem provers are often used by program verification engines. However, the logics implemented by these theorem provers are not a good fit for the program verification domain. In this paper, we have presented a new prover that accurately supports the type of reasoning that program verification engines require. COGENT's strategy is to directly encode input queries into propositional logic. This encoding accurately supports bit operations, structures, unions, pointers and pointer arithmetic, and pays particular attention to the sometimes subtle semantics described in the ANSI-C standard. Our evaluation of COGENT demonstrates that it improves the accuracy of BOOGIE, and both the performance and accuracy of SLAM. Additionally, COGENT provides concrete counterexamples in the case of failed proofs. To the best of our knowledge, COGENT is the only theorem prover that accurately supports pointer arithmetic, unions, structures and bitvectors and produces concrete counterexamples for a logic suitable for program verification.

References

1. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI. (2002)
2. Barnett, M., DeLine, R., Fahndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. JOT **3** (2004) 27–56
3. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs (2003)
4. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In: IFM. (2004)
5. Ball, T., Cook, B., Lahiri, S.K., Zhang, L.: Zapato: Automatic theorem proving for predicate abstraction refinement. In: CAV. (2004)
6. Henzinger, T.A., Jhala, R., Majumdar, R., Qadeer, S.: Thread modular abstraction refinement. In: CAV, Springer Verlag (2003) 262–274
7. Chaki, S., Clarke, E., Groce, A., Strichman, O.: Predicate abstraction with minimum predicates. In: CHARME. (2003)
8. Ivers, J., Sharygina, N.: Overview of ComFoRT, a model checking reasoning framework. Technical Report CMU/SEI-2004-TN-018, CMU (2004)
9. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: TACAS. (2005) to appear.
10. Stump, A., Barrett, C., Dill, D.: CVC: a cooperating validity checker. In: CAV 02: International Conference on Computer-Aided Verification. (2002) 87–105
11. Cook, B., Kroening, D., Sharygina, N.: Accurate theorem proving for program verification. Technical Report 473, ETH Zurich (2005)
12. Ball, T., Cook, B., Das, S., Rajamani, S.K.: Refining approximations in software predicate abstraction. In: TACAS. (2004)

[1] `http://www.inf.ethz.ch/personal/kroening/cogent/`