Zapato: Automatic theorem proving for predicate abstraction refinement

Thomas Ball[†], Byron Cook[†], Shuvendu K. Lahiri[⋆], and Lintao Zhang[†]

[†] Microsoft Corporation [⋆] Carnegie-Mellon University

Abstract. Counterexample-driven abstraction refinement is an automatic process that produces abstract models of finite and infinite-state systems. When this process is applied to software, an automatic theorem prover for quantifier-free first-order logic helps to determine the feasibility of program paths and to refine the abstraction. In this paper we report on a fast, lightweight, and automatic theorem prover called Zapato which we have built specifically to solve the queries produced during the abstraction refinement process.

1 Introduction

Slam is a symbolic software model checker for the C language which combines predicate abstraction together with counter-example guided abstraction refinement. Slam's overall performance critically depends on the automatic theorem prover that is used during the refinement process. Two of Slam's modules, Newton [1] and Constrain [2], generate many theorem proving queries during their analysis[1]. As an example, when Slam was used to check 30 properties of the Microsoft Windows Parallel Port device driver, Newton and Constrain called the prover 487,716 times. On average, these queries contained 19 unique atoms and 40 instances of Boolean operators per query. In the worst case, one query contained 658 unique atoms. Another query contained 96,691 instances of the Boolean operators. For the scalability of Slam it is paramount that these queries are solved in less than a half a second in the average case.

In this paper we briefly describe Slam's theorem prover — called Zapato. Zapato's overall architecture is similar to Verifun [5]. However, rather than using the Simplex or Fourier & Motzkin arithmetic decision procedures, Zapato uses an algorithm due to Harvey & Stuckey [6] for solving constraint problems over sets of *unit two-variable per inequality* integer terms.

2 Lazy theorem proving based on propositional logic SAT

Zapato takes a query in quantifier-free first-order logic (FOL) and returns `valid` if it can prove the query to be true. In the case that it cannot prove the query true, it returns `invalid`. Note that `invalid` does not actually mean that the query is not valid—only that Zapato could not prove it valid. Zapato operates

[1] C2bp [3] can also be configured to use the theorem prover described in this paper. However, by default, C2bp actually uses a special purpose symbolic theorem prover called FastF which is similar to the work of Lahiri, Bryant & Cook [4] but optimized for speed and not precision.

by negating the FOL query passed to it and then trying to prove unsatisfiability. ZAPATO creates a propositional logic (PL) abstraction of the negated query in which each FOL term is replaced by a fresh representative propositional variable. A mapping between terms and propositional variables is maintained throughout the execution of the algorithm.

An incremental PL SAT-solver is used to determine if the propositional abstraction is satisfiable. In the case that it is unsatisfiable, the query is determined to be valid. In the case that the abstraction is satisfiable, the model found is converted to FOL (via a reverse application of the mapping) and passed to a FOL SAT-solver for conjuncts. This model is often *partial*—meaning that variables which can be of either Boolean value in the model are typically eliminated by the SAT-solver in a post-backtracking analysis. Since these values are not important in the model they can be ignored when determining the satisfiability of the model in FOL.

If the input FOL conjunct solver determines that the found model is satisfiable, then the original query is determined to be invalid. In the case that the set of conjuncts are unsatisfiable, the *core reason* that the model is unsatisfiable is determined. This core reason is a subset of the original conjuncts and is found by visiting the leaves of the proof tree constructed by the FOL conjunct solver. The core reason is then converted to PL using the mapping and inserted into the PL SAT-solver's database of learned clauses. A third possible output from the conjunct solver is a set of case-splits which appear as pairs of conflict causes. This case is due to the way some axioms are instantiated. The loop is repeated until the PL SAT-solver returns *unsatisfiable* or the FOL solver returns *satisfiable*.

ZAPATO's FOL conjunct solver uses Nelson & Oppen's method of combining decision procedures with equality sharing. Currently we are using this technique to combine congruence closure for uninterpreted functions with Harvey & Stuckey's decision procedure. Each of the procedures must produce the reasoning behind their decisions in order to allow for the construction of proofs. ZAPATO's FOL solver also implements several axioms that express facts about C expressions with pointers by searching the set of input conjuncts and adding additional conjuncts.

2.1 Harvey & Stuckey's Decision Procedure

Harvey & Stuckey's procedure is a complete algorithm with polynomial time complexity that can be used to solve conjuncted integer constraints. In this procedure each input constraint is expected to be of the form $ax + by \leq d$ where x and y are variables, d is an integer and a and b must be elements of the set $\{-1, 0, 1\}$. Harvey & Stuckey call this subset the *unit two-variable per inequality* (UTVPI) constraints. Note that ZAPATO uses homogenization and term rewriting to remove terms that contain both arithmetic and uninterpreted function symbols. ZAPATO also rewrites arithmetic terms into canonical form. For example, $x < y$ is converted into $1x + (-1)y \leq -1$.

The algorithm is based on iterative transitive closure: as each conjunct is added into the system, all possible consequences of the input are computed. Several strengthening transition rules on the state of the set of known facts are applied at each conjunct introduction to produce the complete closure. In the case that the conjuncts are unsatisfiable, a conjunct and its negation will eventually exist in the set of facts. In the case that no contradiction is found, the constraints must be satisfiable.

An interesting aspect to note about Harvey & Stuckey's algorithm is that it can be used to prove the unsatisfiability of sets of constraints which may include non-UTVPI terms. For example, it can prove the unsatisfiability of

$$n > k' + n' \wedge x < y' \wedge v + 5 < l \wedge y' < z \wedge \neg(x < z) \wedge k < v + z$$

In these cases we can simply prune away the conjuncts that are not UTVPI terms because the non-UTVPI constraints (such as $k < v + z$) may not factor into the reason for unsatisfiability. The above expression is unsatisfiable because the following is unsatisfiable: $x < y' \wedge y' < z \wedge \neg(x < z)$. This allows us to solve many queries that may appear at first to be non-UTVPI.

The logic supported by Zapato dictates that Slam's predicates are always UTVPI constraints with the possibility of uninterpreted function symbols. This is due to the fact that Slam's Newton module only adds new predicates in the case that validity is proved by the theorem prover. At first this may sound disastrously limiting. However, it is not. Both Newton and Constrain can refine the abstraction if they can find *any* spurious transitions in the path that they are examining. Paths are often spurious for multiple reasons. So long as at least one of those reasons can be proved valid using Zapato, the refinement algorithm will make progress. In the worst case, Zapato's limited arithmetic support could cause Slam to report a false bug. In the 10 months that we have been using UTVPI in Zapato we have only seen this occur once while model checking real device drivers.

The completeness of Harvey & Stuckey's procedure allows us to prove the validity of queries other automatic provers struggle with. Consider, for example, the validity of $x < 3 \wedge x > 1 \Rightarrow x = 2$. Because Verifun uses a decision procedure for the real numbers the only way that this can be proved is through the use of special purpose heuristics or with additional axioms.

3 Experimental results

To evaluate the effectiveness of Zapato we used Slam to check 82 safety properties of 28 Windows OS device drivers and recorded statistics about the theorem prover queries. In total, Slam passed 1,748,580 queries to Zapato. Note that Slam caches the interface to the prover. These 1,748,580 queries are the cases in which a pre-computed result could not be found in the cache. Of these queries, 148,509 were proved valid by Zapato. Zapato resolved the remaining queries as invalid. 607,251 queries caused Zapato, at least on one occasion, to invoke Harvey & Stuckey's decision procedure on a set of conjuncts with a non-UTVPI term. Of those 607,251 queries, Zapato was still able to prove 62,163 valid.

To compare Harvey & Stuckey's algorithm versus a more traditional arithmetic decision procedure we also compared the functional result of ZAPATO to SIMPLIFY on the benchmark queries. SIMPLIFY was able to prove 248 queries valid that ZAPATO could not. ZAPATO proved 560 queries valid that SIMPLIFY could not. Because the provers support different logics, a performance comparison is difficult to analyze. But generally, as also reported by Flanagan *et al.* [5], the new lazy PL SAT-based approach is faster. In the case of the hardest invalid query from the benchmarks, ZAPATO's runtime was 8 seconds while SIMPLIFY's was 10 seconds. Both SIMPLIFY and ZAPATO were able to prove the hardest valid query in 3 seconds.

4 Conclusions

ZAPATO's architecture is based on the prover VERIFUN. What makes ZAPATO unique is that it does not use Fourier & Motzkin or Simplex as its decision procedure for arithmetic. Instead, it uses a decision procedure proposed by Harvey & Stuckey which has better worst-case asymptotic complexity and a completeness result for a useful subset of arithmetic terms over the integers. The expressiveness of this subset appears to be sufficient for the verification of Windows device drivers using SLAM. We are currently integrating the Simplex decision procedure into ZAPATO, which we envision could be optionally combined with Harvey & Stuckey's algorithm in applications where full integer linear arithmetic is required.

Acknowledgements

The authors would like to thank Cormac Flanagan, Rajeev Joshi, Sava Krstic, Rustan Leino, Ken McMillan, and Sriram Rajamani for their suggestions related to this work.

References

1. Ball, T., Rajamani, S.K.: Generating abstract explanations of spurious counterexamples in C programs. Technical Report MSR-TR-2002-09, Microsoft Research (2002)
2. Ball, T., Cook, B., Das, S., Rajamani, S.K.: Refining approximations in software predicate abstraction. In: TACAS 04: Tools and Algorithms for Construction and Analysis of Systems, Springer-Verlag (2004)
3. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: PLDI 01: Programming Language Design and Implementation, ACM (2001) 203–213
4. Lahiri, S.K., Bryant, R.E., Cook, B.: A symbolic approach to predicate abstraction. In: CAV 03: International Conference on Computer-Aided Verification. (2003) 141–153
5. Flanagan, C., Joshi, R., Ou, X., Saxe, J.B.: Theorem proving using lazy proof explication. In: CAV 03: International Conference on Computer-Aided Verification. (2003) 355–367
6. Harvey, W., Stuckey, P.: A unit two variable per inequality integer constraint solver for constraint logic programming. In: Australian Computer Science Conference (Australian Computer Science Communications). (1997) 102–111