Scalable Program Analysis through Proof Caching (Doctoral Symposium)

Andrea Aquino
Università della Svizzera italiana (USI) - Faculty of Informatics
Via Giuseppe Buffi 13, 6900 Lugano, Switzerland
andrea.aquino@usi.ch

ABSTRACT

Despite the remarkable advances attained by the SMT community in the last decade, solving complex formulas still represents the main bottleneck to the scalability of program analysis techniques.

Recent research work has shown that formulas generated during program analysis recur, and such redundancy can be captured and exploited by means of caching frameworks to avoid repeating complex queries to solvers. Although current approaches show that reusing formulas syntactically can indeed reduce the impact of SMT solvers on program analysis, they still suffer from being logic-dependent, and performing poorly on huge sets of heterogenous formulas.

The core idea of our approach is to go beyond merely syntactical caching frameworks by designing a caching framework that is able to reuse proofs instead of formulas. In fact, even formulas that are syntactically different can share solutions. We aim to study the recurrence of proofs across heterogeneous formulas, and to define a technique to efficiently retrieve such proofs. We plan to exploit a suitable distance function that measures the amount of proofs shared by two formulas to allow the efficient retrieval of candidate proofs within a potentially large space of proofs. In this paper, we present the problem, draft the core idea, discuss the early results and present our research plans.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model cheking*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Logics of programs*

General Terms

Performance, Theory, Verification

Keywords

Caching, constraint solving, satisfiability

1. RESEARCH PROBLEM

Program analysis techniques have been employed since the early seventies to investigate the correctness of programs. Techniques such as *Symbolic Execution* and *Model Checking* are successfully exploited to reveal failures in industrial software systems, such as wireless sensor networks and operating systems, and to feedback precise information on their nature [3, 6, 8]. Many important industries, like Microsoft [4, 10], IBM [2] and NASA [12] rely on program analysis techniques to test their products.

Program analyzers get in input the model of a program and generate a set of formulas that belong to a given theory and encapsulate the actual or intended behavior of that program. Each formula is solved by means of an SMT solver to determine whether a given property holds on the program under test or not.

The logics involved in this process are computationally complex, the core problem of determining the satisfiability of a formula is in fact either NP-complete or undecidable depending on the logic in use. State of the art program analyzers may fail when their underlying solvers do not produce an answer within a reasonable amount of time. Because of this and despite the remarkable advances of the SMT community, solving complex formulas still represents a main bottleneck to the scalability of program analysis techniques.

The recent work of both Cadar et al. [5] and Visser et al. [13] show that formulas generated during program analysis recur within the analysis of a given program as well as across the analysis of different programs. They thus suggest the use of caching frameworks to store formulas together with their solutions during the analysis of a program, and reuse this information to determine the satisfiability of recurrent formulas: a target formula is syntactically compared with others the satisfiability of which is known; if a match is found, it is possible to infer the satisfiability of the target.

Despite the potential of this approach, these caching frameworks still suffer from many limitations. First of all, they are logic-dependent, thus their applicability is limited to program analysis techniques based on the logics they address. Moreover, relying on syntactical matching, they perform poorly on heterogenous formulas. Our research tackles the problem of designing a new caching framework to overcome such limitations.

State of the art approaches rely on the syntactical equivalence between formulas to reuse proofs. However proofs can be reused not only between syntactically equivalent but also between heterogeneous formulas. Our research aims to define a novel technique to identify formulas that, albeit heterogeneous, may share proofs. In a nutshell, the main intuition our work is based on is that it is possible to reuse proofs instead of formulas, allowing inferring the satisfiability of a formula from the proof of a completely different formula.

Consider for instance the formula $f : x + y \neq 0$ that is proved satisfiable by the model $[x = 1, y = 1]$, and the formula $g : 2x - y < 3$ the satisfiability of which has to be determined. Despite the fact that these formulas are different, since the model of f satisfies g, we can deduce that g is also satisfiable. Likewise, consider the formula $f : x > 1 \wedge x \neq 9 \wedge x < 0$ which is proved unsatisfiable by the unsat-core $[x < 0, x > 1]$, and the formula $g : x < 0 \wedge x > 1 \wedge x + k \neq 0 \wedge k = 1$. Since the unsat-core of f is contained in g we can deduce that g is also unsatisfiable.

Testing whether a model satisfies (or an unsat core is contained in) a given formula is reasonably inexpensive. Nonetheless, when dealing with billions of models and unsat-cores, testing all of them on a target formula is impractical.

We claim that the problem of retrieving a proof for a target formula from a potentially large set of proofs can be solved efficiently by defining a notion of distance between formulas, that is, a *distance function*. Intuitively, this distance function should approximate the inverse of the amount of proofs shared by two formulas. Formulas that share all or many proofs should be close, while formulas that share few or no proofs shall be far. Given a properly defined distance function, we can determine the satisfiability of a target formula trying to reuse only the proofs of the formulas that are closer to it. The details of this approach are discussed in Section 4.

While defining the perfect distance function, that is, the function that precisely identifies the inverse of the amount of proofs shared by any pair of formulas, may be very difficult if not impossible, we argue that it is possible to define a reasonable approximation of this function to be exploited in our context.

We base our work on two research hypothesis: (i) proofs recur much more often than formulas, (ii) reusing proofs can lead to a sensible speed up in the application of program analysis techniques.

With respect to the first hypothesis, the reader should notice that proofs recur at least as much as formulas within the analysis of any given program. In fact, if a formula f can be syntactically reused to infer the satisfiability of another formula g, i.e., f and g are exactly the same formula, then either the model of f satisfies g or the unsat-core of f is contained in g. This strengthens our confidence in the first research hypothesis being true.

To validate the second hypothesis we conducted a broad set of preliminary experiments that confirmed the applicability of the approach, in line with our expectations.

The major contribution of our research will be the design of a new caching framework able to reuse proofs instead of formulas to infer the satisfiability value of new formulas. This includes the definition of a reasonable abstraction to store and manipulate formulas and proofs, and the design of a set of effective distance functions to identify reusable proofs quickly.

2. BACKGROUND AND RELATED WORK

The idea of caching formulas to speed up the application of program analysis techniques has been investigated only recently.

Cadar et al. proposed an efficient symbolic executor for LLVM bitcode called KLEE based on the QF_AUFBV logic [5]. This symbolic executor includes a caching framework tuned for that specific logic. Such caching framework performs basic simplifications on formulas and can determine the satisfiability value of a target formula by a syntactically equivalent formula solved in the past. Using a modified version of the UBTree data-structure [11], it is also able to infer the satisfiability value of a target formula from its satisfiable super-formulas and unsatisfiable sub-formulas.

Visser et al. describe another approach: Green [13], a syntactical caching framework for formulas from few specific logics, namely bounded linear integer arithmetics and string arithmetics. Green can be configured to apply different simplification algorithms on formulas. Moreover, it can perform preliminary variable renaming thus enabling the caching of formulas from many programs. It relies on the Redis [7] in-memory repository to store its constraints and to perform quick lookups.

We have recently proposed a caching framework targeting formulas from a specific subset of the QF_LIA logic [1]. We identified a suitable abstraction to represent these formulas and a canonical form for such representation that support both powerful simplifications on formulas and the inference of the satisfiability value of a target formula from its satisfiable super-formulas and unsatisfiable sub-formulas.

While these works provide evidence that the approach of caching formulas is indeed feasible and can mitigate the scalability limitations of program analysis due to SMT solvers, they all target one or few logics and are not suitable in the case of syntactically heterogenous formulas.

Our approach aims to overcome these limitations by approaching the problem from a different perspective. Current caching frameworks reuse formulas syntactically; we argue that it is possible to reuse their proofs instead. In this way, the satisfiability value of a formula can be inferred from the proof of a completely different formula and, since proofs are a concept shared by all logics, the approach is also logic-independent.

3. RESEARCH QUESTIONS

In this thesis we plan to reduce the impact of constraint solving on program analysis by developing an effective and logic-independent caching framework able to reuse proofs instead of formulas. We intend to address this problem by exploiting the intuition that proofs recur more often than formulas. In detail, we plan to address the following research questions:

Q1 How frequently do formulas generated by means of program analysis recur?
Previous work show that formulas recur often, our plan is to reinforce such evidence.

Q2 How frequently do the proofs of such formulas recur? do they recur more frequently than formulas?
Proofs recur at least as much as formulas. We plan to investigate whether they do recur more often or not.

Q3 Can proofs for a given formula be retrieved efficiently?
When dealing with millions of proofs even a simple linear search is infeasible. Our goal is to design a technique to search proofs in logarithmic time.

4. APPROACH AND CHALLENGES

In the first part of our research we performed a preliminary study aiming to assess the research questions detailed in Section 3. We have studied the frequency of recurrence of formulas and proofs generated during program analysis, and defined a technique to determine the satisfiability of a given formula by reusing the proofs of formulas solved in the past. Given a target formula, our technique produces either a proof of satisfiability, i.e., a model, or a proof of unsatisfiability, i.e., an unsat-core, for that formula. The technique determines the satisfiability value of a target formula searching in a repository of pairs of the form ⟨formula, proof⟩. The repository is populated incrementally by solving formulas the proofs of which cannot be retrieved.

As anticipated in Section 1, the core element of our technique is a function expressing a notion of distance between formulas. Intuitively, two formulas shall be close to each other if they share many solutions, and be far otherwise. Using a suitable implementation of this abstract notion of distance, we can query our repository to retrieve only the proofs that are more likely shared by a target formula. In this way we expect to speed up considerably the search process.

Once defined a distance function, simply calculating the distance of each formula in the repository from the target formula and selecting only the formulas that are closer to it lead to an unacceptable complexity, being in general linear with respect to the size of the repository.

We solve this problem by observing that the formulas in the repository and the distance function form a metric space, and pairing each formula in the repository with its distance from a given formula that we elect as the reference point of the space. In this way, given a target formula, we can identify the formulas that are closer to it in logarithmic time in two steps.

We first calculate the distance d of the target formula from the reference point, and we query the repository to retrieve the entries the distances from the reference point of which are closer to d.

Since the repository is sorted, this step can be performed efficiently but it might retrieve formulas which are not very close to the target as well. To counter this problem, we calculate the distance of any entry retrieved during this first step and the target formula, and we consider only the candidates that are actually closer to the target.

We check whether any retrieved model (respectively, unsat-core) satisfies (respectively is contained in) the target formula. If it is the case, we report a *cache hit* and immediately return the result to the end user. If no match is found, we report *cache miss*, we feed the formula to an SMT solver, store the result in the repository and return it to the end user.

The detailed design and implementation of the technique raises several challenges. A first challenge consists in defining a set of distance functions to be used by our technique. There is a clear trade-off between the complexity, and thus the effectiveness, of such functions and the efficiency of the approach. Our intuition is that more complex distance functions lead to more chances of reuse but require more time to be calculated.

A second challenge consists in finding a suitable representation for formulas and proofs. Since our main concerns are efficiency and compatibility with program analysis, this representation should be both light-weight and standardized.

A last challenge consists in identifying an efficient data structure for the repository. Since we expect to store an impressive number of formulas and proofs, the repository structure should introduce a small per-entry overhead. Moreover, since the main goal of our technique is to find a given proof out of billions guiding the search by means of a distance function, the repository structure should enable efficient searches by proximity to a given value.

5. RESEARCH STATUS

We tackled the challenges described in Section 4 by defining a set of suitable distance functions and implementing a prototype of our technique. The prototype gets in input formulas in the SMT-lib v2 format.[1] This format has been adopted by the most popular constraint solvers and effectively represents a de facto standard for representing formulas from many theories.

To ease the test of models and unsat-cores our prototype also renames the variables of the formulas with fresh names according to their lexicographical order. Moreover, whenever possible, it applies logic-dependent simplifications like the elimination of equalities and the normalization of inequalities.

We designed the prototype repository as two separate structures. The first one stores satisfiable formulas together with their models, the other one unsatisfiable formulas together with their unsat-cores. Both structures are implemented as multisets sorted by distance from a given reference point, i.e., the formula *true* for the former, the formula *false* for the latter. This way, the retrieval of the closest models and unsat-cores from the repository can be performed by means of a k-nearest-neighbours search in logarithmic time.

The prototype is parametric with respect to the distance function. In the experiments conducted so far, we used a modified version of the *Levenshtein* metric, that takes in input two formulas and returns the minimum number of editing operations needed to transform the string representation of one of them into the string representation of the other. An editing operations is either the deletion, the addition or the modification of a character. Although this function poorly approximates the notion of distance defined in Section 1, it is both simple and quite effective in practice.

In the case of a cache miss our prototype feeds the formula to the Microsoft Z3 SMT solver [9], the result is stored in the relevant structure depending on its satisfiability value and returned to the end user.

We evaluated our technique using our prototype to cache formulas generated by means of symbolic execution. All the experiments detailed below were performed using the modified *Levenshtein* metric as the distance function. During the search of each single target formula we retrieved the 50 formulas closer to the target (according to their distance from the reference point) and we retained the 10 formulas actually closer to it.

In the first experiment we compared the effectiveness of our approach (measured as the *reuse rate*, that is, the percentage of formulas in a given dataset it is able to identify as reusable) with the state of the art caching framework implemented in KLEE. We extracted the formulas generated during the analysis of seven popular sorting algorithms (*bubblesort*, *selectionsort*, *heapsort*, *mergesort*, *quicksort*, *shellsort* and *insertionsort*) using the KLEE symbolic executor. Then,

[1]http://www.smt-lib.org

we calculated the amount of formulas that can be reused by the cache implemented in KLEE and with our approach when starting with an empty repository and incrementally storing formulas on cache misses. These algorithms were chosen since, although written in completely different ways, they share the same functionality; moreover, the logic used to analyze them, that is QF_AUFBV, is arguably complex and thus an interesting challenge for any caching framework. Our approach is much more effective than KLEE, being able to reuse almost 3 times the amount of formulas KLEE can reuse in the worst case (boosting its reuse rate from 26.25% to 73.68%), and overwhelming it in the best one.

In the second experiment, we compared our approach with merely syntactical caching frameworks. We generated thousands of random formulas from the QF_LIA logic, with variable coefficients and free coefficients varying in the interval $[-1, 1]$ and a fixed number of clauses and variables (ranging from 1 to 10). We clustered them by size (expressed in terms of the number of clauses and variables) and for each cluster we calculated the percentage of formulas that can be reused both relying on string matching only and with our approach.

While the reuse rate achieved by syntactical means quickly drops to zero when the size of formulas increases, our approach is much more resilient. In fact in the best case, the merely syntactical approach can identify the 53.17% of the formulas as reusable, while our approach boosts this reuse rate to 95.71%. In the worst case, the merely syntactical approach is able to identify 2.90% of reuse opportunities, while our approach still scores a reuse rate of 95.91%.

In the last experiment we tested our approach on formulas generated during the analysis of real world programs, i.e., the Coreutils of the Linux kernel. We analyzed all the 100 Coreutils programs by means of KLEE and focused only on the formulas that were missed by its cache, that is, the formulas on which the cache of KLEE scores a reuse rate of 0%. On this formulas, our approach scores on average a reuse rate of 57.38%. This suggests that our technique can largely improve the state of the art also when applied to real world programs. Our prototype is on average 30% faster than state of the art SMT solvers on the datasets we analyzed.

These experiments confirm our intuition that proofs can indeed be reused, and that reusing proofs can be much more effective than reusing formulas syntactically.

6. FUTURE WORK

The results obtained so far with the simple *Levenshtein* metric are encouraging. We identified several core research directions that we plan to explore in the second part of the PhD.

First of all, the *Levenshtein* metric is logic-independent but is indeed a poor approximation of the notion of distance we defined. Its effectiveness is likely due to the fact that we perform formula simplification prior to search and thus the edit distance of two formulas is close to their semantic distance. We plan to define and test many other logic-independent and logic-specific distance functions to identify a good candidate that can go beyond the limits of the *Levenshtein* metric.

Second, the validation has been carried out on few relatively small case studies. We plan to validate our idea experimentally, by applying our approach tuned with the different selected distance functions to a broader set of formulas extracted from complex programs. We will try to determine the efficacy of the selected metrics and the advantages and the limitations of our approach both in terms of effectiveness (reuse rate) and performance (running time and complexity). We plan to compare our approach both with state of the art solvers and caching frameworks.

7. REFERENCES

[1] A. Aquino, F. Bianchi, M. Chen, G. Denaro, and M. Pezzè. Reusing constraint proofs in program analysis. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2015.

[2] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Transactions on Software Engineering*, 2010.

[3] R. S. Boyer, B. Elspas, and K. N. Levitt. Select – a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*. ACM, 1975.

[4] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 2000.

[5] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2008.

[6] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: Preliminary assessment. In *Proceedings of the International Conference on Software Engineering*. ACM, 2011.

[7] J. L. Carlson. *Redis in Action*. Manning Publications Co., 2013.

[8] L. A. Clarke. A program testing system. In *Proceedings of the 1976 Annual Conference*. ACM, 1976.

[9] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, 2008.

[10] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 2012.

[11] J. Hoffmann and J. Koehler. A new method to index and query sets. In *Proceedings of the International Joint Conference on Artificial Intelligence*. Morgan Kaufmann Publishers Inc., 1999.

[12] C. S. Păsăreanu and N. Rungta. Symbolic pathfinder: Symbolic execution of java bytecode. In *Proceedings of the International Conference on Automated Software Engineering*. ACM, 2010.

[13] W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: Reducing, reusing and recycling constraints in program analysis. In *Proceedings of the SIGSOFT International Symposium on the Foundations of Software Engineering*. ACM, 2012.