Reusing Constraint Proofs for Scalable Program Analysis

Meixian Chen
University of Lugano, Switzerland
meixian.chen@usi.ch

ABSTRACT

Despite the recent advances of research and technology in the area of automated constraint solvers, constraint solving remains a major bottleneck for the scalability of many techniques for program analysis. Recent studies indicate that this problem can be mitigated by caching the proofs for the constraints that occur repeatedly during the analysis of the same or similar programs, showing preliminary evidence that this can result in significantly reducing the analysis time.

My PhD research draws on this initial results and aims to bring the technology for storing and reusing constraint proofs at an entirely new stage of maturity. We believe that equivalent constraints occur frequently across programs, and aim to turn the problem of solving the constraints into a fast and reliable search over proofs shared among different projects and teams through distributed data stores.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Verification

Keywords

Program analysis, constraint solving, symbolic execution, parallel search

1. RESEARCH PROBLEM

Program analysis techniques have been widely investigated for software verification and testing [1, 9, 8]. Most of these techniques rely on constraint solving to decide the satisfiability of formulas generated during the analysis [2]. Symbolic execution is a notable example of a program analysis that uses constraint solvers to check the feasibility of program paths and generate test inputs to execute those paths [11].

ISSTA'14, July 21–25, 2014, San Jose, CA, USA
Copyright 2014 ACM 978-1-4503-2645-2/14/07...$15.00
http://dx.doi.org/10.1145/2610384.2631831

Constraint solving techniques have seen extraordinary progresses in the last decade, and modern solvers, like Z3 [6], are very powerful and fast in dealing with constraints in specific theories. Despite these advances, constraint solving still remains the main bottleneck to the design of scalable program analyses, due to the decidability and complexity problems that underlie the solving procedures for many relevant classes of constraints, and the complexity of the formulas generated when analysing large programs. For example, solving quantifier-free linear constraints of integer number is an NP-complete problem, and solving polynomial constraints is an undecidable problem. To address these problems, different solvers apply different strategies, for example based on term rewriting or approximation techniques, to cope with different categories of problems. In general, we may expect that the analysis of a program may require the use of several (as opposed to only one) constraint solvers, since the single solvers can succeed or fail to solve specific (types of) formulas that arise during the analysis.

Recent studies indicate that many constraints recur frequently during the analysis of the same or similar programs, and that strategies for reusing the proofs of the already solved constraints can significantly improve performance. Cadar et al. have shown that the analysis time can be significantly reduced by reusing the results of constraints queries maintained in a local cache [4]. Visser et al. engineer a constraints caching component, built on top on an in-memory database, to provide initial evidence that linear constraints recur during the analysis of programs with similar functionality [13].

My PhD research draws on these initial results and aims to bring the technology for storing and reusing constraint proofs at an entirely new stage of maturity. The ultimate goal of my work is to speed up scalable program analysis; it depends on two main research hypotheses that equivalent constraints occur frequently across programs and it is possible to build a fast search engine over massive amounts of constraints. Providing evidence of the validity of these hypotheses passes through the achievement of a set of challenging intermediate milestones.

A main challenge is to be able to reveal the equivalence of the constraints that come from different programs, though they can have very different representation and internal structure. For example, different programs are likely to produce constraints that, though possibly equivalent, come with different names of the variables, different ordering of the clauses, different positions of the variables in a clause and mutually subsumed set of clauses. These issues are only partially dealt by the current proposals on the caching and reuse of the constraints [4, 13]. Thus, part of the contribution of my PhD will be the study of advanced algorithms for checking the equivalence of constraints. These algorithms must work with constraints in different theories.

Next, to fully benefit from reusing the proofs contributed by different projects and teams, I envision the use of a distributed storage, and a fast search procedure based on parallel algorithms and a suitable indexing of the storage. I aim to provide a constraint search engine (deployed over the distributed storage and embodying advanced equivalence checking algorithms) that can be faster, as for the average response time, than the constraint solvers. By enabling the reuse of the proofs generated over time by different solvers, in the context of different analyses and out of different programs, this constraint search engine will provide a fast and powerful means for any program analysis that relies on constraint solving. I am currently investigating MapReduce as a platform to define the search over distributed Hadoop repositories of constraints [7, 12].

The main technical goal of my research is to provide evidence that the same constraints recur during the analysis of different programs. Achieving this goal will enable me to investigate this research hypothesis. Equipped with a proper abstract representation of the constraints, practical algorithms for checking their equivalency, and an efficient search engine over a distributed storage of constraints, I will conduct extensive experiments to provide empirical evidence that constraints recur across programs, and that our approach is indeed feasible and successful to improve the scalability of program analyses that rely on constraint solving.

2. BACKGROUND AND RELATED WORK

The idea of caching and reusing constraint proofs through the analysis of a program has been recently proposed and investigated by some research proposals related to symbolic execution.

Some authors extend symbolic execution to the sharing of (sub)constraints among the program paths, as a way to optimize the analysis of a set of incremental versions of a program [14, 10]. The extended symbolic executors maintain information about the (path condition) constraints met during the analysis, and reuse the constraints to identify common prefixes and symmetries of the symbolic execution trees built across multiple analyses of a set of incremental versions of a program. Their results show that the analysis time can be largely reduced by checking only the new/modified paths and the new/modified parts of the formulas instead of all paths and the whole formulas.

Cadar et al. designed KLEE, a symbolic executor to generate high-coverage test cases for software, and experimented with caching and reusing constraints to improve the performance of the analysis [4]. KLEE identifies the independent (sub)formulas within the path constraints generated during symbolic execution, caches the proofs that the solver provides for these formulas, and reuses these proofs for the formulas that are met multiple times during the symbolic execution. The initial data reported in the paper indicates that reusing constraints can save up to 40% of the analysis cost.

Visser et al. studied the amount of constraint proofs reused across different programs with similar functionality [13] . To increase the chances of reusing the proofs, their platform, called *Green*, exploits the renaming of the variables of the constraints and identifies the independent subformulas within the constraints. Their preliminary results indicate that constraints recur across programs and projects.

In my PhD research, I aim to extend the ability to identify equivalent constraints beyond the renaming of the variables and the identification independent sub-formulas (as in KLEE and Green). For example, I am currently considering the hypothesis that the constraints can contain the same clauses in different orders, and the possibility that the same variables occur in different orders (other than different names) within the clauses. Furthermore, the current results available with KLEE and Green focus on linear constraints only. I am also working on extending the equivalence checking to non linear constraints, by designing an abstract representation of the constraints that can simplify the equivalency checking for different classes of constraints, including for example polynomial ones. I also plan to go beyond the simple idea of using local cache, like the UBTree string matching engine used in KLEE or the in-memory database used in Green, investigating the viability of a distributed storage, to enable the sharing of massive sets of constraints between different programs, projects and teams.

3. GOALS, CHALLENGES, APPROACH

My PhD research aims to improve the scalability of program analysis by fostering the reuse of constraint proofs on a large-scale. This will be grounded on new technology to enable a cooperative distributed platform where massive amounts of proofs can be contributed and shared across different projects and teams. Specifically, my PhD addresses the following research goals:

G1 Design efficient algorithms, well grounded in theoretical results, to check for the equivalence between constraints in different theories.

G2 Produce convincing evidence that equivalent constraints occur frequently across programs, and that the ability of reusing the proofs across the recurring constraints is a key factor to achieve scalability in program analysis.

G3 Enable a distributed storage to handle constraint proofs on a large scale. I aim to take advantage of both distributed architectures and parallel computations to meet the performance requirements of the program analysis techniques that are the perspective users of the proofs.

Achieving these goals is a challenge in many respects. The perspective solution depends on the ability of identifying recurring constraints, and on how well we can exploit distributed platforms and parallel computations to store and search through the constraints. In particular, my PhD research will address the following challenges:

C1. Equivalence checking: We need to investigate efficient algorithms to check the equivalence between constraints, and to understand when we can reuse the available proofs. When dealing with constraints from different programs, homomorphic constraints can be represented in heterogeneous ways due to different naming of the variables and different orders of both the clauses within the constraints and the variables within the clauses. The problem of deciding the equivalency between heterogeneous constraints can be reduced from solving the graph isomorphism problem, which is theoretically hard. Furthermore, for constraints in conjunctive form, we are interested in identifying if a proof is available for any unsatisfiable subset or satisfiable superset of clauses. When handling sub/super proofs, the equivalency between constraints can be reduced to sub graph isomorphism, which is an even more complicated problem.

I aim to address this challenge by proposing suitable canonic forms of the constraints, such that checking the equivalence between constraints can reduce (as much as possible) to comparing the characteristics of the corresponding canonic forms. I will study abstract representations of the constraints that facilitate efficient transformations of the formulas to the canonic forms, and the identification of the constraints matched by some sub/super proofs.

C2. Classes of constraints: Constraints from general programs come in different theories and at different levels of complexity. It is unlikely that a single abstract representation or canonic form can suite all constraints in general. Perspectively, the types of representation will differ for constraints in different theories. For example, to represent polynomial constraints we must handle the exponents of the variables that occur in different monomial terms and clauses, other then the position of the variables that we record for a linear constraint. Trigonometric constraints may require yet more complicated representations, and so forth.

In the initial part of my PhD, I focused on representing the linear constraints, which are very frequent in program analysis, using matrices to represent the constraints and calculus on matrices to manipulate the representation. I plan to extend the representation to other classes of constraints incrementally, focusing on polynomials as next step.

C3. Distributed storage: Handling massive amounts of proofs contributed from different projects and teams calls for using a distributed storage and parallel computation to optimize the operations over the storage. The operations to be supported include recording the new proofs incrementally over time, checking for internal consistency of the datastore when adding new proofs, and search as well as the computation of indices for fast search.

My approach is to build a prototype of the solution according to a leading edge platform for distributed management of data and parallel computation, and studying the pros and cons of the platform for the specific problem of handling the constraint proofs. Currently I am investigating the Hadoop distributed filesystem to manage a datastore of constraint proofs, and the MapReduce paradigm to design the operations on the storage as parallel algorithms [7].

C4. Fast search: A specific challenge is to enable a fast search over the massive amounts of proofs handled on the distributed storage. The search of an available proof should be quicker than re-producing the proof from scratch with a constraint solver.

A perspective solution is to design suitable indices to speed up the search. The indices have to represent the key features of the constraints, support the searching of sub and super proofs, and be easy to update when adding proofs to the datastore. I plan to investigate the use of MapReduce to compute the indices, as suggested by the approach of Google to indexing the documents in the Google search.

C5. Experiments: Another main challenge is to design experiments to evaluate my PhD work. My goal is to provide a distributed constraint solving environment for different program analysis techniques, when these are applied by different projects and teams. Thus, for evaluating my work, I need an experimental platform populated with billions of proofs and reference program analysis techniques.

I plan to use existing symbolic executors [4, 3, 10, 11]) to generate constraints and populate the storage with their proofs. I will start by investigating whether and to what extent constraint proofs recur during the analysis of evolving and mutated programs. Both cases entail the analysis for different versions of a program, which most likely share many proofs by construction. In the last year of my PhD, I will take advantage of my advances in the technical side to conduct extensive experiments on the constraints that recur across different programs. I will refer to publicly available open source projects to retrieve suitable experimental subjects.

4. CURRENT STATUS AND PLANS

4.1 Coping with Linear Constraints

In the first part of my PhD, I investigated the feasibility of the approach focusing on linear constraints, and investigated the main element of the approach: abstract representation of constraints, equivalence checking and parallel search, referring to MapReduce and Hadoop as a prototyping platform.

I defined an abstract canonical form of constraints based on matrices of the coefficients, and designed an algorithm to transform linear constraints to this canonic form. The check for equivalency of the constraints in canonical form is reduced to a simple comparison. The algorithm transform a constraint into its canonical form by iteratively permuting the rows and the columns of the matrices. I have demonstrated that the algorithm terminates and converges to the same matrix when applied to equivalent constraints. The details of the algorithm and the related proofs are available as a technical report [5]. An alternative algorithm built on top of software graph algorithms, which solves constraints equivalency problem via graph isomorphism problem, is under investigating.

The canonical form supports parallel search of constraints. This algorithm is implemented in a prototype tool that stores and searches proofs of linear constraints. The prototype stores the constraints as a relational representation in Hadoop, and implements the search algorithm according to the MapReduce parallel computation paradigm. The implementation handles also the proof of constraints that depend on unsatisfiable subsets or satisfiable supersets of clauses for which there already exists a stored proof. The prototype includes a normaliser that transforms the constraints from heterogeneous, tool-specific representations to the canonical form, handling arithmetic simplifications, and identifying the independent sub-constraints that can be checked separately.

The distributed search algorithm based on MapReduce includes two main steps. In a first step, the ”Map” phase, we process the nodes of the distributed file system in parallel to retrieve all constraint clauses that match the ones of the constraint that the user is searching for. In the second step, the ”Reduce phase”, we group the clauses that belong to the same formula and send the groups of clauses to a set of parallel ”Reduce” jobs that check for the equivalency. If we succeed in identifying an equivalent constraint in the storage, we return the associated proof.

Preliminary experiments with a set of constraints derived by using state-of-art symbolic executors indicate that the approach can indeed quickly identify equivalent constraints.

4.2 Future Plans

In the next part of my Ph.D, I plan to (1) define abstract representations and equivalence checking mechanisms for non-linear constraints, (2) investigate enhanced mechanisms to handle distributed data stores of constraint proofs with advanced techniques of parallelism, and (3) provide empirical evidence to demonstrate the feasibility and of the approach.

Dealing with non-linear constraints is the next main challenge. I plan to define abstract representations and canonic forms to overcome the heterogeneity of constraints that come from different programs, thus reducing the check for equivalence to comparing canonic forms. So far, I have collected preliminary data that this approach works for linear constraints, and I now extending these ideas to non-linear constraints. For example, a possible direction for extending the current relational representation and equivalency checking algorithm to polynomial constraints can be (1) representing the polynomial constraints in form $L(V_i) \wedge (\bigwedge V_i = M_i)$, where L is a linear constraint over a set of variables V_i and each V_i represents a monomial M_i, (2) use the matrix-based abstraction to represent the linear relationships between the monomials M_i, and store the specific monomials in a separate table of the data store, and (3) extend the algorithm for equivalence checking to account for the presence of the monomials in the formulas.

The success of this work depends on the ability to handle massive sets of constraint proofs on a large scale by means of distributed and parallel technologies. So far, I have applied the MapReduce paradigm and some related technology, my plans are to better exploit the characteristics of this paradigm to improve the efficiency of the search. The ability of parallelizing the search depends on the representation of the constraints, and defining an optimal abstraction of constraints to improve the parallel search is one of the major challenge and objective of my work. I am currently working on designing an indexing mechanism based on the equivalence characteristics of the constraints in the datastore. The goal of such index is to move the complexity of the computation to the maintenance of the index, while being able to answer very quickly the user queries based on the information in the index. Computing and maintaining the index requires efficient parallel algorithms to be executed off-line with respect to the user queries.

To show the validity of the approach I will need to experiment with enormous amounts of constraints. Investigating the recurrence of equivalent constraints in program analysis and generating a data store large enough to prove the scalability of the approach is a main challenge as discussed in the next subsection.

4.3 Evaluation Plan

To demonstrate the feasibility of my approach, I must estimate the extent to which constraint proofs recur during program analysis, and measure the efficiency of my approach in presence of enormous amount of proofs.

I plan to set up an experimental platform populated with billions of constraint proofs generated from the analysis of different programs, and measure the performance characteristics of the platform under different conditions of use.

The success of my PhD project will provide a distributed constraints solving platform for constraint solving users, and will open the road to scalable program analysis with increased potential for industrial exploitation.

5. REFERENCES

[1] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*. ACM, 2008.

[2] E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013.

[3] P. Braione, G. Denaro, and M. Pezzè. Enhancing symbolic execution with built-in term rewriting and constrained lazy initialization. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013.

[4] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2008.

[5] M. Chen. Recognizing equivalent systems of linear constraints via transformation in canonic form. http://www.people.usi.ch/chenm/matching.pdf, 2014.

[6] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, 2008.

[7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 2008.

[8] P. Godefroid. Test Generation Using Symbolic Execution. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2012.

[9] R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys*, 2009.

[10] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2011.

[11] C. S. Păsăreanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. *International Journal on Software Tools for Technology Transfer*, 2009.

[12] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, 2010.

[13] W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: Reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012.

[14] G. Yang, C. S. Păsăreanu, and S. Khurshid. Memoized symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 2012.