Optimizing Constraint Solving to Better Support Symbolic Execution

Ikpeme Erete and Alessandro Orso
College of Computing, Georgia Institute of Technology
Atlanta, Georgia
{*ikpeme,orso*}*@cc.gatech.edu*

***Abstract*—Constraint solving is an integral part of symbolic execution, as most symbolic execution techniques rely heavily on an underlying constraint solver. In fact, the performance of the constraint solver used by a symbolic execution technique can considerably affect its overall performance. Unfortunately, constraint solvers are mostly used in a black-box fashion within symbolic execution, without leveraging any of the contextual and domain information available. Because constraint solvers are optimized for specific kinds of constraints and heavily based on heuristics, this leaves on the table many opportunities for optimizing the solvers' performance. To address this problem, we propose a novel optimization strategy that uses domain and contextual information to optimize the performance of constraint solvers during symbolic execution. We also present a study in which we assess the effectiveness of our and other related strategies when used within dynamic symbolic execution performed on real software. Our results are encouraging; they show that optimizing constraints based on domain and contextual information can improve the efficiency and effectiveness of constraint solving and ultimately benefit symbolic execution.**

***Keywords*-symbolic execution; constraint solving;**

I. Introduction

Testing is one of the most commonly used techniques to identify faults in a software system. Although software testing cannot guarantee the absence of faults in the code, if suitably performed it can provide some confidence in the correct behavior of that code. One of the main problems in software testing is the generation of inputs to exercise the code under test. Because manual generation of inputs is extremely labor intensive, both researchers and practitioners have investigated ways to automate or semi-automate this task. In fact, the last few decades have witnessed the definition of countless test input generation techniques (*e.g.,* [2], [4], [8]).

Among these techniques, one popular approach is random testing, which randomly selects values from the input domain of a program (*e.g.,* [4]). The main advantage of random testing over manual testing is that it can quickly generate a large number of inputs. Its main drawback is that the generated test inputs tend to be shallow and ultimately result in inadequate testing.

As an alternative to random (and manual) testing, researchers have proposed various test input generation techniques based on symbolic execution. Symbolic execution is not a novel technique, as it was introduced more than 30 years ago by King [7]; however, in recent years, there has been renewed interest in symbolic execution techniques due to the tremendous growth in computational power of the average machine and the availability of increasingly powerful decision procedures. Because symbolic execution relies heavily on constraint-solving technologies, the efficiency of constraints solvers is of paramount importance in this context. One important characteristic of constraint solvers is that they are optimized for specific kinds of constraints and heavily based on heuristics. Therefore, providing contextual information and domain knowledge to the solvers is likely to considerably improve their performance.

Unfortunately, however, most symbolic execution techniques use constraint solvers in a purely black-box fashion, by feeding the solvers constraints to solve without providing them with any other contextual or domain related information. Recently, a few researchers have started to use some domain-based constraint optimizations (*e.g.,* [5], [8]), which represents an important first step in the right direction. However, the effectiveness of these optimizations is unclear. It is also unclear what other optimizations could be used, and ultimately, what could be the benefits of a tighter integration between symbolic execution and constraint solving.

The goal of this paper is to investigate these issues and help provide an answer to these questions. To achieve this goal, we (1) propose a novel optimization based on dynamic symbolic execution, (2) discuss other commonly used optimizations presented in the literature, and (3) perform an extensive empirical study in which we analyze the effect of these new and existing optimizations. In our study, we target two of todays' most popular constraint solvers: CVC3 [1] and Z3 [3]. As data for the study, we use more than 5,000 path conditions that we collected by performing dynamic symbolic execution on three real software subjects.

We believe that the work presented in this paper represents a first important step towards a better integration of symbolic execution and constraint solving techniques by providing the following main contributions:

- Highlight the problems related to the use of constraint solvers as black boxes within symbolic execution.
- Propose a novel constraint-optimization technique based on domain restriction that can improve dynamic symbolic execution.
- Perform an extensive empirical evaluation that provides evidence of the effectiveness of constraint optimization techniques and motivates further research in the area.
- Makes publicly available a large set of real constraints and related infrastructure that can support research in this and other related areas (see http://www.cc.gatech.edu/~ikpeme/software/).

II. Background and Motivating Example

Symbolic execution techniques execute a program with symbolic inputs and try to cover all possible paths in the program [7]. For each path, symbolic execution updates the symbolic state of the program according to the semantics

```
function foo(int a, int b, int c, int d) {
 1. if (c > a)
 2.   int e = d + 10
 3.   if (b > 5)
 4.       //do something
 5.   else if (a < e)
 6.       if (b < c)
 7.           //do something
 8.       else
 9.          //do something
10.   else
11.      //do something
12. return
}
```

Figure 1. Simple code example to illustrate symbolic execution.

of the instructions being executed and keeps track of the constraints that the inputs must satisfy for that path to be followed—*the path condition (PC)*. At the end of the execution of a path, the corresponding PC is fed to a constraint solver. (The constraint solver may also be invoked while executing a path to simplify constraints and detect infeasible paths.) If the solver is able to find a solution for the PC, that solution represents the concrete inputs that lead to the execution of the path being considered. The extraordinary growth in the computational power available on the average user machine and the availability of increasingly powerful decision procedures in recent years have spawned a renewed interest in the use of symbolic execution for test input generation. In particular, one recent variation of symbolic execution called dynamic symbolic execution is becoming increasingly popular (*e.g.,* [2], [5], [8]). For this reason, in this work, we focus on this variation of symbolic execution.

Dynamic symbolic execution (DSE) runs a program using concrete and symbolic inputs at the same time, so that the concrete execution drives the symbolic execution along a specific path that is guaranteed to be feasible. While it executes the path, DSE keeps track of the symbolic state and the PC for that path, which consists of a conjunctive set of constraints, one for each branch traversed. When done, DSE negates the last constraint in the PC that corresponds to a branch not yet covered (for simplicity, we do not consider alternative exploration policies) and invokes a constraint solver to obtain a solution for the modified PC, which we call PC'. We call the negated constraint, which represents the only difference between PC and PC', *target constraint*, and the symbolic variables involved in such constraint *target (symbolic) variables*. If the solver returns a solution for PC' (*i.e.,* a set of inputs that cause the program to follow the path identified by PC'), DSE executes that path and continues the process until either all branches are covered, some other coverage goal is achieved, or a time threshold is reached.

To illustrate DSE, consider the code snippet in Figure 1 and assume that the concrete inputs for the program are initially $a = 4$, $b = 5$, $c = 6$, and $d = 1$. We call the corresponding symbolic inputs a_0, b_0, c_0, and d_0. When DSE executes the program using these concrete values, the execution follows the path $\langle 1, 2, 3, 5, 6, 7, 12\rangle$, and the corresponding PC is $(c_0 > a_0) \wedge (b_0 <= 5) \wedge (a_0 < d_0 + 10) \wedge (b_0 < c_0)$, which corresponds to the conjunction of the predicates for branches $1T$, $3F$, $5T$, and $6T$. DSE would then negate the last constraint, $(b_0 < c_0)$, obtain PC'=$(c_0 > a_0) \wedge (b_0 <= 5) \wedge (a_0 < d_0 + 10) \wedge (b_0 >= c_0)$, and pass PC' to a constraint solver. (In this case, the target constraint is $(b_0 >= c_0)$, and the target symbolic variables are b_0 and c_0.) The constraint solver would then return a possible solution, such as $a_0 = 4$, $b_0 = 5$, $c_0 = 5$, and $d_0 = 1$, that are the input values that satisfy PC'.

Our intuition is that this approach, by asking the solver to find a solution for PC' without leveraging the fact that there is a known solution for PC, misses the opportunity to optimize the solvers' performance. More generally, DSE and other symbolic-execution techniques do not take advantage of any domain knowledge (*e.g.,* the fact that the constraints are generated by a program with specific characteristics and properties) or context information (*e.g.,* the existence of a solution for a very similar set of constraints). Our overall goal is to understand how symbolic execution can be properly integrated to take advantage of these opportunities for optimization. As a first step towards this goal, and as a way to gather initial evidence of the correctness of our intuition, we propose a possible optimization strategy for dynamic symbolic execution. We also study the effects of our new and two existing optimizations on two commonly used constraints solvers when used in the context of DSE.

III. Constraint Optimizations

We present our new constraint optimization technique and discuss two other optimizations presented in previous work.

A. DomainReduce Optimization

Our novel optimization strategy, called DomainReduce, restricts the domain of the constraints to be solved by eliminating irrelevant or potentially irrelevant constraints based on dynamic information. As discussed above, our optimization is specifically targeted to improve constraint solving in the context of DSE. DomainReduce relies on two main observations. (The first observation has also been used in previous work (*e.g.,* by Sen and colleagues [8]). The second one is, to the best of our knowledge, a novel contribution.)

The *first observation* is that, within DSE, constraints that are not dependent on the target constraint (*i.e.,* the one being negated) cannot affect the solution of the modified path condition, PC'. Two constraints c_a and c_b in a set of constraints C are dependent if at least one of the following conditions hold:

1) *Direct dependency*: $vars(c_a) \cap vars(c_b) \neq 0$ where $vars(c)$ represents all symbolic variables in c.
2) *Indirect dependency*: There exists a subset of constraints $\{c_1, ..., c_n\}$ in C, such that (1) c_a and c_1 are dependent, (2) c_i and c_{i+1} (for $i = 1, ..., n-1$) are dependent, and (3) c_n and c_b are dependent.

(Intuitively, c_a and c_b are dependent if the solution of one can affect the solution of the other, which happens if they share one or more variables directly or indirectly.) We also define two variables v_a and v_b as *directly* (resp.,

indirectly) *dependent (symbolic) variables* if they appear in two constraints that are directly (resp., indirectly) dependent. By definition, the solution of all the constraints in PC' that are not dependent on the target constraint are not affected by the solution to that constraint. Therefore, these independent constraints identify a subset of inputs whose values can be "reused" and do not need to be determined by the solver. For our example in Section II, for instance, if PC'=$(c_0 > a_0) \wedge (b_0 <= 5) \wedge (a_0 >= d_0 + 10)$, then our target symbolic variables are a_0 and d_0, and constraint $(b_0 <= 5)$ is independent—the value of b_0 can be safely replaced with that of the corresponding concrete input (*i.e.,* 5). PC' can therefore be simplified to $(c_0 > a_0) \wedge (a_0 >= d_0 + 10)$. For more realistic constraints, this simplification can result in dramatic savings in terms of number of constraints that the solver must process.

The *second observation* is that restricting the domain for PC' may help the constraint solver to find a solution faster than when considering the complete input domain. We propose to invoke the solver first by fixing the value of all target symbolic variables in PC' but one using their concrete values. If the solver can find a solution for this modified PC', this is also a solution for the original PC'; such solution consists of the value found by the solver for the one symbolic variable considered and the original input values for all other symbolic values. If the solver cannot find a solution, we try a different symbolic variable and eventually expand the domain by adding one variable to the set of symbolic variables whose value is not fixed. We continue by adding one variable at a time until either the solver returns a solution, or we have considered all of the target symbolic variables. This technique leverages a tradeoff related to the size of the input domain considered. In general, the smaller the domain, the faster the solver can find a solution, if it exists, but the lower the chances for a solution to exist. Conversely, a larger domain increases the chances for a solution to exist, but can also dramatically increase the computational cost of finding such solution. As our empirical evaluation in Section IV shows, in many cases the additional speed clearly outweighs the reduction in the domain size.

To get a better understanding of the effect of the optimization and of the tradeoffs involved, we defined two variants of the DomainReduce technique: DomainReduce with dependencies and DomainReduce without dependencies.

DomainReduce with dependencies: In this version of the approach, we consider as variables to be passed to the solver not only the target symbolic variables selected, but also their directly or indirectly dependent variables. Figure 2 shows the DomainReduce with dependencies algorithm. First, the algorithm iteratively forms sets of a particular size, starting from size one, using the target variables. Subsequently, the corresponding dependencies for each member of this set are derived, and members with the smallest and largest number of dependencies are determined. (We consider both cases because they represent the two extremes

```
Input: Path condition(PC), Mapping of symbolic variables to concrete values, Solver
Output: result = sat, unsat, or unknown
Constraint cur = getTargetConstraint(index, pc)
List targetsymlist = getSymVars(cur)
result = unknown
for i = 1 to length(targetsymlist) do
    groups[] = formGroupsofX(i, targetsymlist)
    List dependencies = []
    for a = 0 to length(groups) do
        dependencies[a] = getDependencies(groups[a], pc)
    end
    int selectDep[0] = findSmallestDepIndex(dependencies)
    int selectDep[1] = findLargestDepIndex(dependencies)
    for b = 0 to length(selectedDep) do
        List grpDep = dependencies[(selectDep[b])]
        List symbolicVars = join(groups(selectDep[b]), grpDep)
        newPC = modifyPC(symbolicVars,concreteValuesMapping,pc)
        output = callSolver(solver,newPC)
        if output = "sat" then
            return output
        end
        else if output = "unsat" then
            result = output
        end
    end
end
```

Figure 2. DomainReduce with dependencies.

of a spectrum of possible approaches.) The variables in the selected sets determine, by exclusion, which variables in PC' will be replaced with their concrete values (obtained from the inputs that generated PC). The resulting PC' is then sent to the constraint solver. Consider again our example in Section II, for which PC' is $(c_0 > a_0) \wedge (b_0 <= 5) \wedge (a_0 < d_0 + 10) \wedge (b_0 >= c_0)$. In this case, our approach would first pick one of the two target variables, for instance b_0, and ask the constraint solver to solve the modified PC' $(b_0 <= 5) \wedge (b_0 >= 6)$. If the solver could not find a solution for that PC', our algorithm would then consider target variable c_0 and pass to the solver PC' $(c_0 > a_0) \wedge (a_0 < d_0 + 10) \wedge (5 >= c_0)$, where d_0 is also considered because it is indirectly dependent on c_0 through a_0. This is a typical case in which the approach would be beneficial, as the solver would likely be able to find a solution for the first modified PC' quickly and would not require further interactions.

DomainReduce without dependencies: DomainReduce with dependencies accounts for all symbolic variables that might be affected by changes in the values of the target variables selected. To explore the effect of dependencies on the performance of the constraint solver, we also defined a second approach, called DomainReduce without dependencies. This algorithm, depicted in Figure 3, is analogous to the previously described approach, except that it ignores dependencies among variables. For our example, the difference with respect to the behavior of DomainReduce with dependencies is the following: in the second step, when only target variable c_0 is considered, the approach would pass to the solver PC' $(c_0 > 4) \wedge (5 >= c_0)$, where a_0 and b_0 are replaced with their concrete values.

B. Additional Optimizations

Other researchers have defined different strategies for constraint optimization, such as fast unsatisfiability checks [8] and local constraint caching [2]. Although useful, these are general strategies that can be easily incorporated into

```
Input: Path condition(PC), Mapping of symbolic variables to concrete values, Solver
Output: result = sat, unsat, or unknown
Constraint cur = getTargetConstraint(index, pc)
List targetsymlist = getSymVars(cur)
result = unknown
for i = 1 to length(targetsymlist) do
    groups[] = formGroupsofX(i, targetsymlist)
    List allrandomlySelectedGroup = []
    while length(allrandomlySelectedGroup) < length(groups) do
        Found = False
        while ¬Found do
            selectedGroup = randomSelectGroup(groups)
            if selectedGroup ∉ allrandomlySelectedGroup then
                Found = True
                allrandomlySelectedGroup.add(selectedGroup)
            end
        end
        newPC = modifyPC(groups[selectedGroup],concreteValuesMapping,pc)
        output = callSolver(solver,newPC)
        if output = "sat" then
            return output
        end
        else if output = "unsat" then
            result = output
        end
    end
end
```

Figure 3. DomainReduce without dependencies

a constraint solver and do not take into account domain or contextual information. In this paper, we are mostly interested in optimizations that are specifically targeted at improving symbolic execution and, more specifically, DSE. Therefore, in addition to DomainReduce, in our investigation we also consider two optimization strategies presented in related work: incremental solving, by Sen and colleagues [8], and constraint subsumption, by Godefroid and colleagues [5]. *Incremental solving* identifies dependencies between constraints in a set and leverages them to eliminate irrelevant constraints. However, unlike our technique, it does not perform any kind of domain reduction. In our example, PC' obtained using Incremental solving would be $(d_0 < b_0) \wedge (b_0 <= 5) \wedge (d_0 >= c_0)$. Incremental solving can be considered equivalent to the worst case for DomainReduce with dependencies, that is, the case where none of the domain reductions work and all target variables and all of their dependencies must be considered. *Constraint subsumption* identifies cases where a new constraint definitely implies or is definitely implied by another constraint generated from the same instruction and, if so, eliminates the implied constraint. This is especially useful for loops, where this approach can dramatically reduce the number of constraints generated by large numbers of loop iterations.

IV. Empirical Evaluation

Although other researchers have investigated the possibility of optimizing constraint solving within symbolic execution, there is little evidence and understanding of the potential benefits of these optimizations. To address this issue, we performed a study in which we quantitatively analyzed the effectiveness of the three optimizations described in the previous section. More precisely, we investigated whether the use of these optimizations can improve efficiency and effectiveness of constraint solvers in the context of DSE.

A. Study Setup

To conduct our studies, we used three software subjects: HTMLParser (http://sourceforge.net/projects/htmlparser/), XMLParser (http://sourceforge.net/projects/nanoxml/), and K-NN (http://sourceforge.net/projects/weka/). The first two subjects are parsers, whereas the third one is an implementation of the K-Nearest Neighbor machine learning algorithm. We selected these subjects because they are freely available, and most importantly, they can be handled by the DSE framework we used. As a representative implementation of DSE, we used JFuzz [6], which is a dynamic symbolic executor built on top of Java PathFinder (JPF – http://babelfish.arc.nasa.gov/trac/jpf/). We first modified JFuzz, so that it dumped all PC and PC' constraint sets computed during DSE. We then used this modified JFuzz to dynamically execute each subject using 10 different data sets as inputs.

We generated our data sets for HTMLParser using Yahoo!'s random URL generator (http://random.yahoo.com/bin/ryl), obtained the ones for XMLParser from an open xml repository (http://www.cs.washington.edu/research/xmldatasets/www/repository.html), and gathered the ones for K-NN from its distribution. The second column of Table I shows the number of path conditions collected for each subject. As constraint solvers for our study, we considered CVC3 [1] and Z3 [3] because they are popular, actively maintained, and support the SMTLIB standard format (http://www.smt-lib.org/). Finally, we implemented in Python the three optimization strategies considered: DomainReduce, incremental solving, and constraint subsumption.

To gather the data for our study, we used CVC3 and Z3 to find a solution for each PC collected; we fed to the solvers first the PC unoptimized, and then the PC optimized using each of the three approaches considered. For each invocation of each constraint solver, we measured the time used by the solver to return a solution and the outcome: sat, unsat, and unknown. (Note that the time is cumulative in the case of DomainReduce, which may perform more than one invocation of the constraint solver per PC.) The outcome *sat* indicates that the solver found a solution, *unsat* that there is no solution, and *unknown* that the solver timed out before completing its execution. As a time limit for the solvers, we chose ten minutes because we believe it is an adequately long time and yet allows us to perform our data collection in a reasonable overall time. (Note that the time limit used in the STMLIB competition is 20 minutes, which is a comparable time). We also computed the number of target variables and the number of clauses in each set of constraints considered before and after optimization.

We ran our study on 8 dedicated 64-bit quad-core Linux boxes. The data for the different optimizations of a same set of constraints were collected on the same box to avoid problems of bias related to different speeds on different boxes (although all machines had the same configuration).

B. Results and Evaluations

We evaluate our results along two dimensions: number of constraints successfully processed and time required to solve constraints. The first dimension is related to the benefits of the optimizations in terms of allowing the solvers to

Table I
RESULTS OF OUR STUDY IN TERMS OF NUMBER OF CONSTRAINTS SUCCESSFULLY PROCESSED.

Subjects	PCs considered	PCs successfully processed unsat+sat									
		No optimization		**Subsumption**		**Incremental solving**		**DomainReduce with dependencies**		**DomainReduce without dependencies**	
		cvc3	z3	cvc3	z3	cvc3	z3	cvc3	z3	cvc3	z3
HTMLParser	1879	**1879** 43+1836	**1879** 43+1836	**1879** 43+1836	**1879** 43+1836	**1879** 43+1836	**1879** 43+1836	**1879** 43+1836	**1879** 43+1836	**1879** 43+1836	**1879** 43+1836
XMLParser	1881	**473** 49+424	**1881** 49+1832	**473** 49+424	**1881** 49+1832	**1881** 49+1832	**1881** 49+1832	**1881** 49+1832	**1881** 49+1832	**1881** 49+1832	**1881** 49+1832
K-NN	1930	**261** 261+0	**936** 936+0	**261** 261+0	**936** 936+0	**271** 271+0	**937** 937+0	**262** 262+0	**878** 878+0	**111** 0+111	**0** 0+0

process more constraints. The second dimension focuses on the benefits of the optimizations in terms of speeding up the constraint-solving process. Although these two dimensions are related, examining them independently allows us to perform a more thorough analysis of the results.

1) Number of Constraints Successfully Processed: Table I presents the results of our study, for each of the three subjects and two solvers considered, in terms of number of constraints successfully processed. The table is organized as follows: Column 2 (PCs considered) shows the total number of constraint sets considered for each test subject; Column 3 (PCs successfully processed) shows, for each solver and optimization strategy, the number of constraints for which the solver terminated and, underneath, the breakdown of this number between constraints for which it reported that the constraints were unsolvable (*i.e.,* unsat) and solvable (*i.e.,* sat). For example, for XMLparser, 1,881 constraints were considered of which, without optimizations, CVC3 was able to find a solution for 424, reported that they were unsolvable for 49, and timed out for the remaining 1,406.

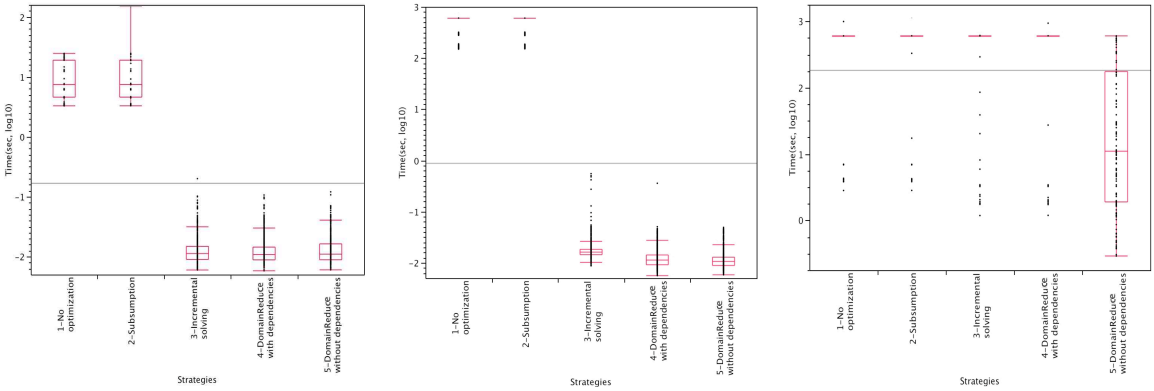
As the table shows, the optimizations are not effective in the case of Z3; either Z3 is able to successfully process all of the constraint sets considered, or it is able to successfully process only a subset of those, and the optimizations do not help. The only exception is incremental solving, which allows Z3 to successfully process one more constraint set for K-NN. It is also worth noting that the use of DomainReduce actually reduces the number of constraint sets that both Z3 and CVC3 can successfully process for K-NN. The reason for this behavior lies in the way DomainReduce operates and in the type of constraints considered for K-NN. Because (1) none of the constraints for K-NN is solvable and (2) in the case of unsat solutions DomainReduce must keep iterating until it tries all possible constraint groups, these constraints represent a worst-case scenario for this optimization strategy. Interestingly, however, in the case of CVC3, all the constraints that DomainReduce without dependencies is able to process are constraints on which all other optimizations time out (see also Section IV-B2).

The situation is quite different for CVC3, where, for two of the three subjects considered, the optimizations are effective in most cases. For K-NN, incremental solving and DomainReduce with dependencies slightly increase the number of constraints successfully processed by CVC3. For XMLParser, these increases are fairly dramatic: incremental solving, DomainReduce with dependencies, and DomainReduce without dependencies all increase the number of constraints successfully processed by CVC3 from 25% to 100% of the constraints considered.

2) Time Required to Solve Constraints: Figures 4 and 5 present, as a box-and-whisker plot, the average time (in log_{10}) spent by each solver to solve the constraints for a test subject when a given optimization strategy was used. As the figures show, overall all of the optimizations but subsumption improved the efficiency of the constraint solvers dramatically in four out of six cases. For both HTMLParser and XMLParser, incremental solving and DomainReduce reduced the runtime of CVC3 by several orders of magnitude on average. And for the same subjects, the optimizations also reduced the runtime of Z3 by one order of magnitude.

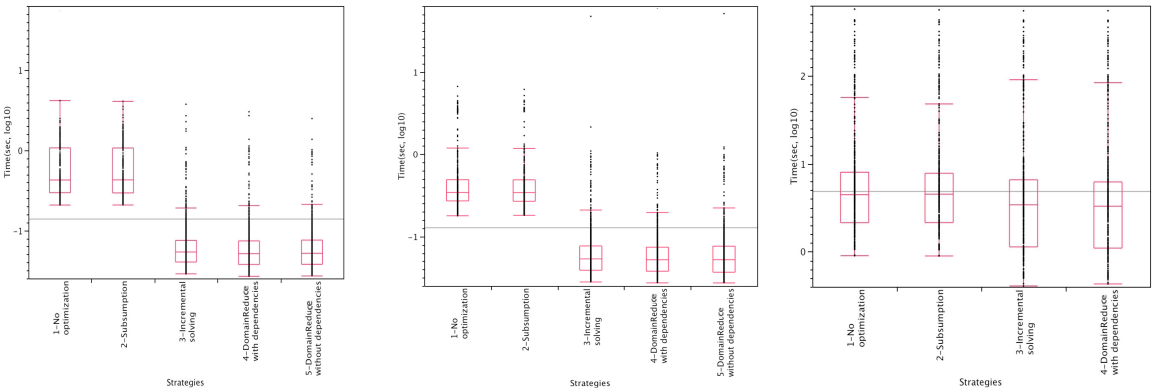
The results for K-NN are, also in this case, quite different from the ones for the other two subjects. For K-NN with Z3, incremental solving and DomainReduce with dependencies also improved the performance of the solver, although to a lesser extent than for HTMLParser and XMLParser. (Note that the results for DomainReduce with dependencies are not presented because it was not able to successfully process any of K-NN's constraints in this case.) For K-NN and CVC3, all but one optimizations provided no benefits. The reason, again, is in the nature of the constraints considered for this subject: in many cases, the constraint solvers timed out while trying to find a solution; in the remaining cases, they returned unsat as a result, but took a long time to compute such result. Interestingly, the only optimization performing well in this case is DomainReduce without dependencies; although it was able to successfully process less constraint sets than the other strategies, it solved those constraint sets two orders of magnitude faster than any of those strategies.

To assess the statistical significance of our results, we performed a two-way Analysis of Variance (ANOVA) test. The analysis substantiated our results in all cases except for K-NN and Z3, where the difference in performance between no optimizations and incremental solving or DomainReduce was not deemed statistically significant. The ANOVA test also revealed that, for all subjects and solvers but K-NN with CVC3, there is no statistical difference between the improvements in performance obtained using incremental solving, DomainReduce with dependencies, and DomainReduce without dependencies. This is because these subjects exhibit either very high or very low coupling among constraints.



(a) HTMLParser using CVC3 solver (b) XMLParser using CVC3 solver (c) K-NN using CVC3 solver

Figure 4. Time taken for each strategy. Time is expressed in log_{10} due to the disparity in results between strategies.



(a) HTMLParser using Z3 solver (b) XMLParser using Z3 solver (c) K-NN using Z3 solver

Figure 5. Time taken for each strategy. Time is expressed in log_{10} due to the disparity in results between strategies.

In other words, either most of the inputs or too few inputs interact with one another, which is the DomainReduce's worst case scenario that makes it equivalent to incremental solving (see Section III-B).

V. Discussion, Conclusion, and Future Work

In this paper, we have argued that symbolic execution and constraint solving should be more tightly integrated. As a first step in this direction, we have presented one new and two existing constraint optimization strategies and assessed whether they can benefit constraint solving and, ultimately, symbolic execution techniques.

Overall, our results are encouraging and motivate further research in this area. Although the use of our and others' optimizations did not always increase the number of constraints that a solver could successfully process, it could in some cases increase such number considerably. Moreover, when efficiency of the constraint solvers is considered, in most cases the optimizations provided dramatic improvements in the time required to process a constraint.

One interesting venue for future work is to investigate in which cases the different optimizations work, so as to be able to select the right approach for the program and type of constraints at hand. Another possible (and related) research direction is the definition of ways to apply several optimizations in parallel (*e.g.,* on a multicore machine), which would allow for trying different strategies at once and stopping as soon as one of the strategies is successful. Finally, future research should investigate other optimization strategies of different nature, such as strategies that leverage domain information (*e.g.,* the structure or properties of the program) to inform and improve constraint solvers.

Acknowledgments

This work was supported in part by NSF awards CCF-0725202 and CCR-0209322 to Georgia Tech.

References

[1] C. Barrett and C. Tinelli. Cvc: A Cooperating Validity Checker. In *Proceedings of CAV 2002*, pages 500– 504, 2002.

[2] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex System Programs. In *In Proceedings of Usenix SOSDI 2008*, pages 209–224, 2008.

[3] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of TACAS 2008*, pages 337– 340, 2008.

[4] J. E. Forrester and B. P. Miller. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In *In Proceedings of the Usenix Windows System Symposium*, pages 59–68, 2000.

[5] P. Godefroid, M. Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of NDSS 2008*, 2008.

[6] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kietzun. JFuzz: A Concolic Whitebox Fuzzer for Java. In *NASA Formal Methods Symposium*, pages 121–125, 2009.

[7] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[8] K. Sen, D. Marinov, and G. Agha. Cute: A Concolic Unit Testing Engine for C. In *In Proceedings of ESEC-FSE 2005*, pages 263–272, 2005.