Heuristically Matching Solution Spaces of Arithmetic Formulas to Efficiently Reuse Solutions

Andrea Aquino[§], Giovanni Denaro[♯], Mauro Pezzè[§♯]

[§] Università della Svizzera italiana (USI)
Via Giuseppe Buffi, 13
Lugano, Switzerland 6900
{andrea.aquino, mauro.pezze}@usi.ch

[♯] University of Milano-Bicocca
Viale Sarca, 336
Milano, Italy 20126
denaro@disco.unimib.it

***Abstract*—Many symbolic program analysis techniques rely on SMT solvers to verify properties of programs. Despite the remarkable progress made in the development of such tools, SMT solvers still represent a main bottleneck to the scalability of these techniques. Recent approaches tackle this bottleneck by reusing solutions of formulas that recur during program analysis, thus reducing the number of queries to SMT solvers. Current approaches only reuse solutions across formulas that are equivalent to, contained in or implied by other formulas, as identified through a set of predefined rules, and cannot reuse solutions across formulas that differ in their structure, even if they share some potentially reusable solutions.**

In this paper, we propose a novel approach that can reuse solutions across formulas that share at least one solution, regardless of their structural resemblance. Our approach exploits a novel heuristic to efficiently identify solutions computed for previously solved formulas and most likely shared by new formulas. The results of an empirical evaluation of our approach on two different logics show that our approach can identify on average more reuse opportunities and is markedly faster than competing approaches.

***Keywords*-SMT-based program analysis; symbolic execution; SMT solvers; solutions reuse**

I. Introduction

Program analysis techniques automatically determine whether properties hold on programs, and find important industrial applications in many relevant domains, such as wireless sensor networks and distributed systems [7], [18], drivers of popular operating systems [3], [10], firmware of medical devices [15], and mission critical software applications [2], [17].

Many program analysis techniques use *Satisfiability Modulo Theories* (SMT) solvers to determine the satisfiability of formulas generated during the analysis. For example *symbolic execution*, a popular program analysis technique, associates the execution paths of a program with formulas, *path conditions*, that represent the constraints that must hold on the inputs of that program to execute those paths. Symbolic execution relies on SMT solvers to determine the satisfiability of the path conditions, and uses this information to identify feasible and infeasible paths.

Despite the remarkable progress in the theory and the maturity of modern solvers, SMT solvers still represent a main bottleneck to the scalability of SMT-based program analysis techniques [6], [13], since the problem of determining the satisfiability of a formula is very hard in general, for instance NP-complete for propositional logic and undecidable for non-linear integer arithmetic.

Recently, Cadar et al. [6], Visser et al. [19], Jia et al. [11] and Aquino et al. [1] have proposed techniques to reduce the number of queries issued by SMT-based program analysis techniques to SMT solvers. These techniques store and reuse solutions[1] of formulas to infer the satisfiability of other formulas. These techniques exploit equivalence-preserving simplifications and syntactical matching rules to identify formulas that are equivalent to, contained in or implied by other formulas, and reuse their solutions. If two formulas can be transformed into exactly the same formula by applying a predefined set of equivalence-preserving simplifications, then the two formulas are equivalent, and the satisfiability of one of them can be inferred from the satisfiability of the other one. If all clauses of a formula in conjunctive normal form are contained in another formula known to be satisfiable, then the former formula is also satisfiable. Similarly, if all clauses of a formula in conjunctive normal form known to be unsatisfiable are contained in another formula, then the latter formula is also unsatisfiable. Yet, if two formulas contain inequalities that differ only in the free coefficient and the comparison operator, then their comparison may lead to infer that one formula implies the other one, and this information can be exploited to infer the satisfiability of formulas implied by formulas known to be satisfiable, or the unsatisfiability of formulas that imply formulas known to be unsatisfiable.

These techniques open new horizons towards the scalability of SMT-based program analysis techniques, but are limited to reuse solutions only across formulas that can be transformed into structurally similar forms, and thus miss many reuse opportunities related to formulas that share solutions but have dissimilar structures. For instance, the formulas

$$x \geq 0 \wedge y \geq 0 \wedge x + y = 0, \text{ and } x = 0 \wedge y = 0,$$

[1] In this paper, we use the term *solution* to refer to either a model of a satisfiable formula or a (possibly minimal) unsatisfiable core of an unsatisfiable formula, as discussed in detail in Section II.

belonging to the quantifier-free linear integer arithmetics, share exactly the same set of solutions, that is, the singleton solution $\{[x = 0, y = 0]\}$, but no current approach can transform them to the same form and thus reuse the solution of one formula to infer the satisfiability of the other one.

This paper presents a novel approach, *Utopia* (Use That Proof Again), based on the core idea of identifying reusable solutions by comparing the solution spaces of the formulas rather than their syntactical structure: the more the solution spaces of two formulas are similar, the higher the probability that a solution computed for one of the formulas can be reused as a solution of the other one.

Utopia heuristically approximates the similarity between the solution spaces of two formulas by (i) measuring the behaviour of each formula with respect to a predefined set of models (assignments of values to variables), a metric that we refer to as the *Sat-delta* value of a formula, and (ii) estimating the similarity of the solution spaces of two formulas as inversely proportional to the distance of their *Sat-delta* values. Given a target formula whose satisfiability has to be determined, *Utopia* computes the *Sat-delta* value of the target formula, and tests the target formula against the solutions of a small set of already-solved formulas with *Sat-delta* values closest to the *Sat-delta* value of the target formula. In this paper we define *Sat-delta* for the integer and real arithmetic logics.

The experimental results obtained on formulas from two different logics (which are discussed in Section III) support the hypothesis that *Sat-delta* well approximates the similarity of the solution spaces of most formulas generated by SMT-based program analysis techniques, including symbolic execution and symbolic model checking.

This paper contributes to the state of the art by (i) proposing a new approach to efficiently reuse solutions of formulas by exploiting their behavioural rather than their structural similarity, and (ii) discussing a set of experimental results that confirm the effectiveness of our approach both in absolute terms and with respect to competing approaches.

The remainder of this paper is organised as follows. Section II describes the *Utopia* approach, and introduces the *Sat-delta* heuristic. Section III describes the experimental results that assess the effectiveness and the efficiency of *Utopia*. Section IV describes the main threats to the validity of the results presented in this paper. Section V compares *Utopia* with the main approaches to mitigate the impact of SMT solvers on SMT-based program analysis techniques. Section VI concludes summarising the main results presented in this paper and outlining our research plans.

II. A Novel Approach to Reuse Solutions

In this section we present *Utopia*, our novel approach to reuse the solutions of some formulas to infer the satisfiability of other formulas. We introduce the terminology, the approach and the *Sat-delta* heuristic at the core of *Utopia*.

A. Terminology

In this paper we use the term *solution* to refer to either the model of a satisfiable formula or the (possibly minimal) unsatisfiable core of an unsatisfiable formula.

The *model* of a satisfiable formula is an assignment of values from a given domain set to the variables of a formula that makes the formula hold true.

An *unsatisfiable core* (in short *unsat-core*) of an unsatisfiable formula in conjunctive normal form is a subset of the clauses of that formula whose conjunction is unsatisfiable. A *minimal unsatisfiable core* of an unsatisfiable formula is an unsat-core of that formula that becomes satisfiable if any of its clauses is removed. For instance, $\{x = 0, x \neq 0\}$, $\{x = 0, x > 0\}$ and $\{x = 0, x \neq 0, x > 0\}$ are the three unsat-cores of the formula $x = 0 \wedge x \neq 0 \wedge x > 0$, the first two of which are minimal.

B. *The* Utopia *Approach*

Differently from current approaches that reuse solutions across formulas with similar syntactical structure, *Utopia* reuses solutions across formulas with similar solution spaces, thus widening the reuse opportunities. *Utopia* measures the similarity of the solution spaces of two formulas by introducing a novel heuristic, *Sat-delta*, that quantifies the behaviour of a formula with respect to a predefined set of models.

Utopia takes as input a formula in conjunctive normal form and identifies either a model that satisfies it or an unsat-core that proves that it is unsatisfiable by searching in two repositories containing solutions of formulas solved in the past. Specifically, these repositories contain satisfiable formulas paired with their models and unsatisfiable formulas paired with their unsat-cores, respectively. The formulas in both repositories are sorted according to their *Sat-delta* value.

Sat-delta quantifies the behaviour of a formula with respect to a predefined set of models, and is based on the intuition that formulas with similar behaviours are likely to have similar solutions spaces. Being sorted according to their *Sat-delta* value, the formulas in the repositories are thus sorted according to the similarity of their solutions spaces allowing for the efficient retrieval of candidates solutions for a target formula. In the remainder of this section we describe *Utopia* referring to the *Sat-delta* heuristic as a black-box. We define *Sat-delta* in the next section.

Algorithm 1 presents the core algorithm of *Utopia*. *Utopia* is parametric with respect to:

- a set A of models to tune the *Sat-delta* heuristic,
- an integer number k_s representing the maximum number of candidate models our approach tries to reuse to solve each formula, and
- an integer number k_u representing the maximum number of candidate unsat-cores our approach tries to reuse to solve each formula.

Utopia relies on an SMT solver (SOLVER) and two repositories (SAT-REPO and UNSAT-REPO) containing formulas paired with their corresponding solutions sorted by their *Sat-delta* value, to solve a target formula (*formula*) as follows:

Algorithm 1 The *Utopia* algorithm

Require:
A: a set of models to tune the *Sat-delta* heuristic.
k_s and k_u: two integers.
SOLVER: an SMT solver.
SAT-REPO and UNSAT-REPO: two repositories containing formulas paired with their corresponding solutions sorted by *Sat-delta*.

```
1: RENAMEVARS(formula)
2: δ ← Sat-delta(formula, A)
3:
4: models ← CLOSEST(SAT-REPO, δ, k_s)
5: for model ∈ models do
6:     if SHARESMODEL(formula, model) then
7:         return model
8:     end if
9: end for
10:
11: unsat-cores ← CLOSEST(UNSAT-REPO, δ, k_u)
12: for core ∈ unsat-cores do
13:     if SHARESUNSATCORE(formula, core) then
14:         return core
15:     end if
16: end for
17:
18: status ← CHECK(SOLVER, formula)
19: if status = SAT then
20:     model ← GETMODEL(SOLVER)
21:     STORE(SAT-REPO, formula, model, δ)
22:     return model
23: else if status = UNSAT then
24:     core ← GETUNSATCORE(SOLVER)
25:     STORE(UNSAT-REPO, formula, core, δ)
26:     return core
27: end if
28:
29: // SOLVER cannot produce a solution.
30: return None
```

Utopia preprocesses *formula* by renaming its variables according to their lexicographical order (line 1 of Algorithm 1). Then, *Utopia* calculates the *Sat-delta* value of the resulting formula (line 2), as discussed in the next section.

Then, it queries the SAT-REPO and the UNSAT-REPO repositories to extract the k_s models and k_u unsat-cores of the formulas whose *Sat-delta* values are the closest to the *Sat-delta* value of *formula* (lines 4 and 11). Since the formulas in both repositories are sorted according to their *Sat-delta* values, this operation can be performed in logarithmic time by means of a k-nearest-neighbours search.

Finally, it checks whether any of these models or unsat-cores is a solution of *formula* (lines 5 and 12). If this is the case (*cache hit*), *Utopia* returns the solution and terminates. If this is not the case (*cache miss*), *Utopia* solves *formula* by means of the SMT solver SOLVER (line 18), stores *formula* and its solution in the relevant repository (lines 21 and 25), returns the solution, and terminates.

Utopia checks whether the model of a formula is a solution of *formula*, by evaluating *formula* on the model,[2] and whether the unsat-core of a formula is a solution of *formula*, by checking whether all the clauses in the unsat-core are syntactically contained in *formula*. Both checks can be performed in linear time with respect to the size of *formula* and the size of the considered models or unsat-cores.

C. *The* Sat-delta *heuristic*

The *Sat-delta* heuristic at the core of our approach is a function that quantifies the behaviour of a formula with respect to a set of models.

Intuitively, given a model, *Utopia* calculates the *Sat-delta* value of a formula by substituting the variables of that formula with the corresponding values specified in the model, and measuring by how much the model fails in satisfying the formula. It follows that the *Sat-delta* value of a formula with respect to a model is zero if that model satisfies the formula, and is a positive number proportional to the distance of the model from the solution space of the formula, if the formula is not satisfied by that model: the further a model is from satisfying a formula, the larger is the corresponding *Sat-delta* value.

Figure 1 presents the *Sat-delta* heuristic and shows how it is calculated on formulas in conjunctive normal form with no negation operators[3].

The *Sat-delta* heuristic quantifies by how much, on average, a given set of models fails in satisfying that formula. As shown in Figure 1a, the *Sat-delta* heuristic is a function that takes as input a formula f and a set A of models, and returns the average *distance from satisfiability* of f from the models in A. Both the accuracy of *Sat-delta* in describing the behaviour of a formula and its cost are directly proportional to the number of models in A.

As shown in Figure 1b, *Sat-delta* calculates the distance from satisfiability of a formula with respect to a single model (*Sat-delta-formula*), as the sum of the individual distances from satisfiability of the clauses of that formula from that model. Since to satisfy a formula a specific model must satisfy all its clauses, and the distance from satisfiability of each clause with respect to a model represents the amount by which that model does not satisfy that clause, their sum is a realistic estimate of the distance of the model from the solution space of the formula as a whole.

As shown in Figure 1c, *Sat-delta* calculates the distance from satisfiability of the clause of a formula with respect to a

[2]If a formula contains variables than do not occur in the model, we extend the model with assignments to default values.

[3]The negation operators of a formula belonging to the quantifier-free linear integer arithmetic logic in conjunctive normal form can be removed by substituting the comparison operators of the negated inequalities with their opposite comparison operators.

$$\textit{Sat-delta}(f, A) = \text{avg}(\{\textit{Sat-delta-formula}(f, a) \mid a \in A\})$$

The *Sat-delta* heuristic computes the average distance from satisfiability of a formula f in conjunctive normal form containing no negation operators from a set A of models. The formula f is a conjunction of clauses of the form $f = \bigwedge c$, each clause $c \in f$ is a disjunction of inequalities of the form $c = \bigvee i$ and each inequality $i \in c$ is in the form $e_1 \odot e_2$ where e_1 and e_2 are algebraic expressions and $\odot$ is a comparison operator in the set $\{\leq, <, =, \neq, >, \geq\}$.

(a) The *Sat-delta* heuristic.

$$\textit{Sat-delta-formula}(f, a) = \sum_{c \in \text{clauses}(f)} \textit{Sat-delta-clause}(c, a)$$

(b) *Sat-delta-formula* computes the distance from satisfiability of a formula f with respect to a single model a.

$$\textit{Sat-delta-clause}(c, a) = \min(\{\textit{Sat-delta-ineq}(i, a) \mid i \in \text{inequalities}(c)\})$$

(c) *Sat-delta-clause* computes the distance from satisfiability of a clause c with respect to a single model a.

$$\textit{Sat-delta-ineq}(e_1 \odot e_2, a) = \begin{cases} 0 & \text{if } a(e_1) \odot a(e_2) \\ |a(e_1) - a(e_2)| & \text{if } \odot \in \{\leq, =, \geq\} \\ |a(e_1) - a(e_2)| + 1 & \text{if } \odot \in \{<, \neq, >\} \end{cases}$$

(d) *Sat-delta-ineq* computes the distance from satisfiability of an inequality of the form $e_1 \odot e_2$ with respect to a single model a.

Fig. 1: The *Sat-delta* heuristic

single model (*Sat-delta-clause*), as the minimal distance from satisfiability of the inequalities of that clause from that model. Since to satisfy a clause, a model must satisfy at least one of its inequalities, a realistic estimate of the amount by which a model does not satisfy a clause is the smallest amount by which it does not satisfy any of its inequalities.

As shown in Figure 1d, *Sat-delta* calculates the distance from satisfiability of an inequality with respect to a single model (*Sat-delta-ineq*), as the smallest non-negative number that must be added or subtracted to the left hand side of the inequality to make the model satisfy it. If the given model satisfies the inequality, the *Sat-delta* value is zero. If the given model does not satisfy the inequality and the comparison operator of the inequality allows it to be satisfied when both sides are set to the same value, then the resulting *Sat-delta* value is the difference of the values obtained by evaluating both sides of the inequality on that model, because by adding that amount to one side the value of the other side is obtained. In case the comparison operator of the inequality does not allow the inequality to be satisfied when both sides are set to the same value, then the resulting *Sat-delta* value is the same of the latter case incremented by one because once the two sides of the inequality are made equal one must still add or subtract something to the left hand side of the inequality to make the model satisfy it. We increment by one because it is the smallest positive integer value that fits our purpose. Let us consider for instance the inequality $2 < 3$; adding one to the left hand side would make both sides equal ($3 < 3$) but would not satisfy the inequality. To make the inequality hold true it is necessary to add more than one to the left hand side, and according to our definition we would add two.

D. Sat-delta*: a proxy to solution space similarity*

To test our hypothesis that formulas with similar *Sat-delta* values may have similar solution spaces, we performed an experiment on a large amount of randomly generated pairs of formulas that include:

- pairs of formulas with identical solution spaces,
- pairs of formulas with overlapping solution spaces,
- pairs of formulas with disjoint solution spaces.

If our hypothesis holds, we expect to measure a very small distance between the *Sat-delta* values of pairs of formulas with identical solution spaces, a larger distance for pairs of formulas that share some, though not all, solutions (that is, formulas with overlapping solution spaces), and an even larger distance for pairs of formulas that share no solutions (that is, formulas with disjoint solution spaces).

We generated 10,000 pairs of formulas for all possible combinations of formula pair type (*identical*, *overlapping* and *disjoint* solutions spaces) and number of variables of the formulas (1, 2, 3, 10 and 100), resulting in 15 datasets composed of 10,000 formula pairs each.

Each formula is generated by constraining each variable in an interval whose extremes are random numbers, thus the solution space of each random formula can be interpreted as a box in a hyperspace. For instance, the solution spaces of the formulas with a single variable can be interpreted as segments on an oriented axis, the solution spaces of the formulas with two variables can be interpreted as rectangles on the cartesian plane, and so on.

We generated the formula pairs with identical solution spaces by producing a random formula and applying to it a set of equivalence-preserving rewriting rules, to sensibly change its structure without altering its meaning. We generated the formula pairs with overlapping solution spaces by producing a random formula and randomly altering its free coefficients in a way that the solution space of the resulting formula partially overlaps the solution space of the original formula (any possible overlap is equally likely). We generated the formula pairs with disjoint solution spaces by selecting a random formula and randomly altering its free coefficients in a way that the solution space of the resulting formula is completely disjoint from the solution space of the original formula but still close to it in the hyperspace.

For each pair of formulas in each dataset, we calculated the distance between the *Sat-delta* values of the formulas. For consistency with the experiments reported in the following section we calculated *Sat-delta* with respect to the following set of models: (i) the model that sets all variables to -10000, (ii) the model that sets all variables to 0, and (iii) the model that sets all variables to 100. Table I reports the average distance we measured for each of our 15 datasets.

TABLE I: Average distance between the *Sat-delta* values of the formulas in each pair in each dataset

#Vars	Identical	Overlapping	Disjoint
1	0	232	956
2	1	439	1959
3	1	660	2900
10	4	2217	9662
100	37	22184	96365

In line with our expectations, no matter the number of variables in the formulas, the distance between the *Sat-delta* value of two formulas is always extremely small if the formulas have identical solution spaces, much larger if the formulas have partially overlapping solution spaces or solution spaces included in one another, and extremely large if the formulas have completely non-overlapping solution spaces.

These results provide some preliminary evidence of the accuracy of *Sat-delta* in distinguishing formulas with similar solution spaces.

In the next section we describe a set of experiments in which we apply the *Utopia* approach, instantiated with the *Sat-delta* heuristic, on datasets of formulas generated by SMT-based program analysis techniques during the analysis of real programs.

III. Evaluation

In this section we discuss the results of a set of experiments to assess the effectiveness and the efficiency of our approach. We performed our experiments on two benchmarks composed of formulas belonging to two logics, the quantifier-free linear integer arithmetic logic and the quantifier-free non-linear real arithmetic logic.

Our experiments address the following research questions:

- RQ_1 (*Effectiveness*) Does *Utopia* identify more reuse opportunities than competing approaches?
- RQ_2 (*Efficiency*) Does *Utopia* outperform modern SMT solvers and competing approaches?

A. Prototype

To evaluate the effectiveness and the efficiency of *Utopia* we implemented a prototype in C++.

Our prototype takes as input a formula and separates it into its mutually independent sub-formulas. Two sub-formulas are mutually independent if they do not have variables in common. The satisfiability of each sub-formula is determined individually, and the results are aggregated to produce a solution of the original formula. This process is known as *formula slicing* [6] and is currently exploited by all competing approaches.

To determine the satisfiability of each individual sub-formula, our prototype calculates its *Sat-delta* value with respect to the following three models:

- the model that sets all variables to -10000,
- the model that sets all variables to 0, and
- the model that sets all variables to 100.

We experimented with many different sets of models to tune *Sat-delta*, in particular with sets composed of one, three, five and ten models but we did not notice significative differences in the results. We chose this group because it produces the best trade-off between reuse opportunities identified by our approach and its efficiency on our benchmarks.

Once our prototype computes the *Sat-delta* value of a target formula, it selects from the two repositories the ten satisfiable and the ten unsatisfiable formulas whose *Sat-delta* values are the closest to the target formula (in other words, our prototype instantiates the algorithm discussed in Section II setting both k_s and k_u to 10).

Finally, our prototype checks whether any of the solutions of the selected formulas is also a solution of the target formula. If it is the case, our prototype reports a *cache hit*. If not, our prototype reports a *cache miss*, calls the Microsoft Z3 SMT solver [8] to produce a solution for the target formula and stores the target formula paired with its solution in the relevant repository.

Our prototype uses Microsoft Z3 version 4.4.2, the latest version available at the time of our experiments, with no timeouts. All formulas considered in our experiments can be solved by Z3 in no more than a few seconds.

In the experiments we invoke Z3 from scratch on every formula due to technical limitations of our current prototype. Our approach can be easily adapted to use the underlying solver incrementally.

B. Experimental Setting

We collected approximately 800,000 formulas belonging to the quantifier-free linear integer arithmetic logic and approximately 30,000 formulas belonging to the quantifier-free non-linear real arithmetic logic in two benchmarks. We produced the first benchmark running the symbolic executors *Crest* [5]

and *JBSE* [14] on 22 C and Java programs. We produced the second benchmark running the model checker *Gk-tail* [4] on 16 Java classes belonging to the *GraphStream* and *Google Guava* libraries. We will refer to these two benchmarks as the *Crest* benchmark and *Gk-tail* benchmark, respectively.

In all our experiments, our prototype starts with empty repositories and considers the target formulas one by one. It incrementally populates the repositories with the formulas (and their corresponding solutions) that it does not manage to solve reusing solutions of other formulas considered in the past. In detail, when *Utopia* does not manage to reuse the solutions in the repositories to solve a formula, it solves that formula with the Microsoft Z3 SMT solver and stores the formula and its solution in the relevant repository, depending on the satisfiability of the formula.

The reader should notice that this experimental procedure may produce different reuse results when formulas are considered in different orders. In our experiments, we considered the formulas in our benchmarks in the order they were produced by the *Crest* and *JBSE* symbolic executors and by the *Gk-tail* model checker, to match a realistic scenario for the application of *Utopia*. To control for possible biases related to considering the formulas in this specific order, we ran our experiments reversing the order of formulas in our benchmarks and rearranging them randomly, and in all cases we did not reveal significant differences in the results.

We treat each program dataset in our benchmarks as a single experiment, clearing the cache between any two experiments.

We performed all the experiments on a MacBook Pro equipped with a 2.3 GHz Intel Core i7 processor and 16 GB of RAM.

C. RQ_1 – Effectiveness

We address the effectiveness of *Utopia* by calculating the amount of formulas in the *Crest* and *Gk-tail* benchmarks that *Utopia* can solve by reusing the solutions of other formulas in the same benchmark. We compare *Utopia* with *Green* [19], *GreenTrie* [11], *Recal* and *Recal+* [1].

We executed our prototype on both benchmarks and measured its *reuse rate*, calculated as the number of the cache hits reported by our prototype on a dataset divided by the total number of formulas in that dataset. The reuse rate effectively represents the percentage of formulas in a dataset that can be solved by our prototype reusing solutions of formulas solved in the past, that is, without calling the SMT solver. Columns *Utopia* in Table II and III report the reuse rates of our prototype for the *Crest* and *Gk-tail* benchmarks, respectively. Columns *Green*, *GreenTrie*, *Recal* and *Recal+* in Table II report the reuse rate of the competing approaches on the *Crest* benchmark. We could not execute the competing approaches on the *Gk-tail* benchmark, since these approaches target formulas belonging to the quantifier-free linear integer arithmetic logic, and do not handle the formulas belonging to the quantifier-free non-linear real arithmetic logic in the *Gk-tail* benchmark.

As a baseline for the evaluation, we use a random selection strategy, which draws ten satisfiable formulas and ten unsatisfiable formulas at random from the repositories and tries to reuse their solutions to determine the satisfiability of each formula. This random selection strategy can be trivially implemented and is extremely fast, thus if the reuse rate of an approach is less than the reuse rate of the random selection strategy, the approach is arguably useless in practice. Columns *Random* in Table II and III report the reuse rates of the random selection strategy for the *Crest* and *Gk-tail* benchmarks, respectively.

We executed an exhaustive search to compute the optimal reuse rates, that is, the best reuse rates that can be achieved by all techniques considering all the solutions available in our benchmarks, to identify the upper bound to the effectiveness of all techniques. Columns *Maximum* in Table II and III report the best achievable reuse rates for the *Crest* and *Gk-tail* benchmarks, respectively.

In Table II, we highlight in red the reuse rates worse than the random selection strategy, and in green the reuse rates better than or equal to all approaches which are the closest to the maximum achievable value. *Utopia* is the only approach that is never worse than the random selection strategy on all datasets, and largely outperforms it on three datasets: (i) the *avl* dataset (by 75%), (ii) the *cdaudio* dataset (by 91%), and (iii) the *treemap* dataset (by 68%). In absolute terms, this corresponds to more than 250,000 solutions that can be reused with *Utopia* and not with a random selection strategy.

The results reported in Columns *Utopia* and *Maximum* of Table II show that *Utopia* is nearly optimal on the *Crest* benchmark. In fact, the reuse rates of *Utopia* are very close to the theoretical maximum. They only differ for a few datasets: (i) the *avl* dataset (by 7%), (ii) the *block* dataset (by 11%), (iii) the *dijkstra* dataset (by 1%), (iv) the *kbfiltr* dataset (by 1%), (v) the *new-tax* dataset (by 17%), and (vi) the *old-tax* dataset (by 21%).

The results reported in columns *Utopia*, *Random* and *Maximum* of Table III show that *Utopia* is nearly optimal also on the *Gk-tail* benchmark, and (often largely) better than the random selection strategy for all datasets. The reuse rates of *Utopia* are more than 10% lower than the optimal reuse rates it could have possibly achieved only for three datasets: (i) the *guava.ArrayListMultimap* dataset (by 17%), (ii) the *guava.ImmutableMultiset* dataset (by 16%), and (iii) the *guava.TreeMultimap* dataset (by 15%). *Utopia* is always at least as good as the random selection strategy on all datasets in the *Gk-tail* benchmark and largely outperforms it on most of them. In the best case, that is, the *graphstream.MultiGraph* dataset, the reuse rate produced by *Utopia* outperforms the one produced by the random selection strategy by 69%. In particular, in the three cases of reuse rate lower than the optimal by more than 10%, *Utopia* largely outperforms the random strategy.

The reuse rates of *Green*, *GreenTrie*, *Recal* and *Recal+* reported in Table II show that: (i) the *Green* approach is not particularly effective on this benchmark, being able to reuse on average only 39% of the formulas in a dataset, (ii) the

TABLE II: Reuse rates of solutions in the *Crest* benchmark

Program	#Formulas	#Sat	#Unsat	Green	GreenTrie	Recal	Recal+	Utopia	Random	Maximum
afs	203	203	0	76%	97%	99%	99%	99%	99%	99%
avl	11161	10420	741	73%	84%	93%	93%	91%	16%	98%
ball	210	210	0	6%	55%	84%	83%	94%	70%	94%
block	505	505	0	33%	56%	35%	39%	49%	14%	60%
cdaudio	55329	47292	8037	16%	65%	99%	99%	99%	8%	99%
collision	6812	1526	5286	76%	97%	99%	99%	99%	72%	99%
dijkstra	85	43	42	0%	48%	49%	51%	68%	56%	69%
diskperf	103505	100000	3505	44%	87%	99%	99%	99%	99%	99%
division	1257	1	1256	0%	99%	0%	97%	99%	99%	99%
floppy	100006	100000	6	46%	93%	99%	99%	99%	99%	99%
grep	100126	100000	126	0%	99%	99%	99%	99%	99%	99%
kbfiltr	188	176	12	11%	51%	91%	96%	95%	95%	96%
knapsack	7651	0	7651	57%	99%	58%	59%	99%	99%	99%
list	876	686	190	87%	96%	87%	91%	96%	79%	96%
multiplication	25217	0	25217	0%	99%	0%	99%	99%	98%	99%
new-tax	55	37	18	36%	82%	36%	62%	65%	65%	82%
old-tax	43	29	14	37%	86%	37%	70%	65%	65%	86%
reverseword	38104	70	38034	99%	99%	99%	99%	99%	99%	99%
swapwords	173	0	173	0%	99%	0%	96%	99%	99%	99%
tcas	13476	292	13184	39%	98%	99%	99%	99%	63%	99%
treemap	332950	170450	162500	96%	99%	99%	99%	99%	31%	99%
wbs	239	143	96	20%	66%	97%	97%	97%	97%	97%
min	43	0	0	0%	48%	0%	39%	49%	8%	60%
average	36281	24186	12094	39%	84%	71%	88%	92%	74%	94%
max	332950	170450	162500	99%	99%	99%	99%	99%	99%	99%

Legend

Better than or equal to all competing approaches
Worse than random

TABLE III: Reuse rates of solutions in the *Gk-tail* benchmark

Program	#Formulas	#Sat	#Unsat	Utopia	Random	Maximum
graphstream.MultiGraph	5703	4778	925	71%	2%	74%
graphstream.SingleGraph	4496	3944	552	67%	2%	69%
guava.ArrayListMultimap	1907	1001	906	48%	8%	65%
guava.ConcurrentHashMultiset	448	206	242	52%	22%	54%
guava.Duration	21	10	11	43%	43%	43%
guava.HashBiMap	1071	897	174	78%	17%	82%
guava.HashMultimap	280	152	128	51%	11%	52%
guava.HashMultiset	34	20	14	44%	44%	44%
guava.ImmutableBiMap	33	13	20	39%	30%	39%
guava.ImmutableListMultimap	30	13	17	20%	13%	20%
guava.ImmutableMultiset	1263	768	495	49%	6%	65%
guava.LinkedHashMultimap	1697	963	734	30%	2%	32%
guava.LinkedHashMultiset	136	66	70	61%	54%	61%
guava.LinkedListMultimap	2597	1932	665	56%	9%	60%
guava.TreeMultimap	7465	5284	2181	35%	2%	50%
guava.TreeMultiset	1389	1018	371	58%	8%	64%
min	21	10	11	20%	2%	20%
average	1786	1317	469	50%	17%	55%
max	7465	5284	2181	78%	54%	82%

Recal approach is consistently better than *Green*, being able to reuse on average 71% of the formulas in a dataset, but is still sensibly outperformed by the other approaches, (iii) the *GreenTrie* approach is extremely effective on this benchmark, being able to reuse on average 84% of the formulas in a dataset and is only slightly outperformed by the *Recal+* approach, for which the average is 88%, (iv) *Utopia* achives the best average reuse rate, being able to reuse on average 92% of the formulas in a dataset.

It is worth to recall that all approaches except *Utopia* perform worse than the random selection strategy on some datasets. In particular, the *Green* approach performs worse than the random selection strategy on 15 datasets, *Recal* on 8 datasets, *GreenTrie* on 7 datasets and *Recal+* on 5 datasets.

In summary, the experiments discussed in this section indicate that the reuse opportunities identified by *Utopia* are on average higher than the ones identified by the state-of-the art approaches based on reusing solutions of formulas.

D. RQ_2 – *Efficiency*

We address the efficiency of *Utopia* by comparing the computation time of *Utopia* with its competing approaches to solve the formulas in our benchmarks. We logged the execution time of *Utopia* and all the competing approaches for all the experiments described in the previous section. We also calculated the execution time of the Microsoft Z3 SMT solver to solve the formulas in each dataset to provide a baseline for the evaluation of the efficiency of *Utopia*.

To minimise the bias due to the execution environment, we performed all measurements on a freshly rebooted machine running no other user-level programs, repeated each measurement ten times, and averaged the results. The differences in the results we measured across these ten runs were negligible (they differed by no more than 5% of the average value we measured).

Table IV and Table V report the result of this experiment for the *Crest* and *Gk-tail* benchmarks, respectively. In table IV, we highlight in red the execution times worse than Z3, and in green the best execution times. In the first case, the approach to reuse solutions is useless, in the second case it is the best out of the considered approaches.

The data reported in Table IV show that *Utopia* is consistently faster than the competing approaches and always faster than Z3 on the *Crest* benchmark. As shown in row *total*, *Utopia* can solve all formulas in the *Crest* benchmark in 633 seconds, approximately twelve times faster than Z3, and almost twice faster than the second best, the *GreenTrie* approach, that solves all formulas in 1046 seconds. All other approaches are consistently slower: *Green* solves all formulas in 3650 seconds, *Recal* in 5992 seconds and *Recal+* in 7254.

Interestingly, the *Recal+* approach, which is one of the best approaches in terms of reuse rate, is also the slowest, merely saving 500 seconds with respect to Z3 on this benchmark.

The data reported in Table V show that *Utopia* is consistently faster than Z3 on the *Gk-tail* benchmark. As shown in row *total*, *Utopia* solves all formulas in this benchmark in 66 seconds, approximately six times faster than Z3, which needs 387 seconds to solve all formulas.

In summary, the experiments discussed in this section indicate that *Utopia* outperforms Z3 and all the competing approaches, being approximately twelve times faster than Z3 in the best case and almost twice faster the second best competing approach *GreenTrie*.

IV. Threats to Validity

Our results may be biased by faults in our prototype implementation. We extensively tested our prototype and verified the correctness of the solutions identified with it for a large set of formulas. To do so, we determined the satisfiability value of these formulas with the Microsoft Z3 SMT solver and checked the consistency of the outcome of our prototype. We also validated a large portion of the solutions identified by our prototype by hand.

Our results may also be biased by the structure of the formulas in our benchmarks. These formulas were produced by analysing 22 C and Java programs with the *Crest* and *JBSE* symbolic executors and by producing dynamic software behavioural models with the *Gk-tail* model checker from 16 Java classes. The formulas in the *Crest* benchmark are large conjunctions of linear inequalities over integer variables, while the formulas in the *Gk-tail* benchmark are implications between disjunctions of inequalities over real variables. These formulas have very similar structures; this might not hold for formulas generated by other SMT-based program analysis techniques.

Our results may be biased by the order in which we considered the formulas in our benchmarks. As reported in Section III we considered the formulas in our benchmarks in the order they were produced by *Crest*, *JBSE* and *Gk-tail* to match a realistic scenario for the application of our approach. We also reversed the order of formulas in our benchmarks and rearranged them randomly, without revealing any significative difference in the results.

Finally, in the experimental evaluation of our approach, we did not compare our approach with the caching frameworks implemented in *Klee*. We could not compare *Utopia* with *Klee* because the caching frameworks implemented in *Klee* target formulas belonging to the quantifier-free theory of bit-vectors and bit-vector arrays and thus cannot handle the formulas in any of our benchmarks, and our prototype cannot currently handle formulas containing bit-vectors and bit-vector arrays.

V. Related Work

The problem of mitigating the impact of SMT solvers on the cost of SMT-based program analysis techniques has been addressed in many different ways, including (i) using multiple solvers to mitigate the weaknesses of individual solvers, (ii) complementing SMT solvers with external optimisations, and more recently (iii) reducing the number of queries issued by SMT-based program analysis techniques to SMT solvers.

A first thread of work has focused on using multiple SMT solvers at the same time to mitigate the weaknesses

TABLE IV: Running times (in seconds) on the *Crest* benchmark

Program	Z3	Green	GreenTrie	Recal	Recal+	Utopia
afs	1.55	0.79	0.31	1.20	1.30	0.13
avl	57.27	27.15	19.72	68.99	73.51	9.14
ball	0.45	1.37	0.63	0.31	0.37	0.07
block	1.26	2.48	1.65	1.87	2.02	0.83
cdaudio	327.05	339.47	154.72	144.86	155.80	31.67
collision	19.37	12.58	2.49	12.00	12.24	2.01
dijkstra	1.33	2.10	2.08	1.66	1.64	0.60
diskperf	767.24	470.43	150.40	454.03	561.56	104.23
division	40.09	37.79	23.58	46.84	63.05	5.72
floppy	520.20	396.20	92.64	280.10	308.86	50.20
grep	3505.48	1569.77	277.26	2190.59	2589.96	206.46
kbfiltr	0.48	1.36	0.59	0.22	0.23	0.05
knapsack	648.92	332.02	222.01	1958.61	2532.03	60.58
list	1.72	1.16	0.26	0.85	0.86	0.17
multiplication	162.26	189.03	7.53	90.36	68.42	12.12
new-tax	0.10	0.34	0.06	0.10	0.08	0.04
old-tax	0.08	0.25	0.03	0.08	0.05	0.03
reverseword	148.65	10.49	4.21	33.22	38.19	6.68
swapwords	5.41	5.76	2.91	5.92	7.71	0.84
tcas	56.54	68.94	5.51	21.10	24.43	5.88
treemap	1458.83	179.73	77.31	679.58	811.38	135.53
wbs	0.69	1.66	0.55	0.26	0.30	0.06
min	0.08	0.25	0.03	0.08	0.05	0.03
average	351.13	165.95	47.57	272.40	329.73	28.77
max	3505.48	1569.77	277.26	2190.59	2589.96	206.46
total	7724.95	3650.87	1046.46	5992.76	7254.00	633.03

Legend

Better than or equal to all competing approaches
Worse than Z3

TABLE V: Running times (in seconds) on the *Gk-tail* benchmark

Program	Z3	Utopia
graphstream.MultiGraph	125.25	12.74
graphstream.SingleGraph	106.24	10.85
guava.ArrayListMultimap	9.36	2.88
guava.ConcurrentHashMultiset	1.80	0.75
guava.Duration	0.04	0.03
guava.HashBiMap	2.13	0.65
guava.HashMultimap	1.34	0.47
guava.HashMultiset	0.05	0.04
guava.ImmutableBiMap	0.14	0.07
guava.ImmutableListMultimap	0.13	0.09
guava.ImmutableMultiset	7.41	2.45
guava.LinkedHashMultimap	25.09	7.80
guava.LinkedHashMultiset	0.31	0.17
guava.LinkedListMultimap	16.59	4.18
guava.TreeMultimap	81.17	21.20
guava.TreeMultiset	10.11	1.82
min	0.04	0.03
average	24.20	4.14
max	125.25	21.20
total	387.16	66.17

of individual solvers. Palikareva et al. have observed that modern SMT solvers have different strengths and that, for most queries, it is impossible to tell in advance which solver will perform better [16]. They have proposed a framework, which runs many solvers in parallel on a given formula, and returns the solution produced by the first terminating solver. Since converting formulas to strings, shipping these strings to SMT solvers and having the solvers parse them back produces an unacceptable overhead, Palikareva et al. suggest to use macros to concurrently assert a formula to many solvers by means of their APIs. This framework has been integrated in the *Klee* symbolic executor largely improving its scalability.

A different thread of work has focused on complementing SMT solvers with external optimisations. Erete and Orso have proposed a technique that exploits contextual information to restricts the domain of formulas generated during the dynamic symbolic execution of a program to eliminate potentially irrelevant constraints [9].

Li et al. exploit machine learning approaches to satisfy complex formulas that cannot be solved with modern solvers [12]. They drive the machine learning process with a measure of formula dissatisfaction, which is closely related to our heuristic *Sat-delta* although used differently.

A more recent thread of work has focused on reducing the number of queries issued by SMT-based program analysis techniques to SMT solvers by storing and reusing solutions of formulas. This thread of work has led to the design of four techniques that are closely related to our work and that we survey in further detail in the remainder of this section.

Cadar et al. have designed *Klee* [6], a symbolic executor that includes two caching frameworks that target formulas belonging to the quantifier-free theory of bit-vectors and bit-vector arrays. These two caching frameworks are called the *branch cache* and the *counterexample cache*. The branch cache simply "remembers" formulas and their solutions. The counterexample cache is used to determine whether a target formula contains or is contained in other formulas solved in the past. If the target formula is contained in a satisfiable (resp. unsatisfiable) formula, then it is also satisfiable (resp. unsatisfiable). If the target formula contains a satisfiable formula, the solution of such formula is also checked on the target formula potentially proving it satisfiable.

Visser et al. have designed *Green* [19], a caching framework that targets formulas belonging to the quantifier-free linear integer arithmetic logic. Given a target formula, this framework simplifies it exploiting a predefined set of simplification rules. Then, it checks whether the resulting formula is contained in a database of other formulas solved in the past. If the target formula matches a satisfiable (resp. unsatisfiable) formula in the database, then it is also satisfiable (resp. unsatisfiable).

Jia et al. have designed *GreenTrie* [11], an extension of the *Green* caching framework that also targets formulas belonging to the quantifier-free linear integer arithmetic logic. *GreenTrie* can identify some particular cases of logical implication between two formulas. Given a target formula, this framework checks whether it implies or is implied by other formulas solved in the past. If the target formula is implied by a satisfiable formula, then it is also satisfiable. If the target formula implies an unsatisfiable formula, the it is also unsatisfiable.

Aquino et al. have designed *Recal* [1], a caching framework that targets formulas belonging to the quantifier-free linear integer arithmetic logic. Given a target formula, this framework simplifies it exploiting a predefined set of simplification rules, transforms it into a matrix and applies to it a canonicalisation algorithm to make the order of its variables and clauses irrelevant. Then, it checks whether the resulting formula is contained in a database of other formulas solved in the past. Similarly to *GreenTrie*, another version of this framework (*Recal+*) developed by the same authors can also identify some particular cases of logical implication between two formulas.

The *Utopia* approach described in this paper is substantially different from all these approaches. Instead of trying to identify formulas that are equivalent to, contained in or implied by other formulas by exploiting a predefined set of syntactical rules, our approach exploits the *Sat-delta* heuristic to identify the formulas that more likely share solutions with a given target formula.

The *counterexample cache* implemented in *Klee* is the only existing framework that includes a component that checks whether the solution of a formula is shared by one of its super-formulas by effectively evaluating such solution. Our approach generalises this idea reusing solutions even across formulas with little or no structural resemblance that may or may not be contained in one another.

VI. Conclusion

In this paper, we presented *Utopia*, a new approach to reduce the impact of SMT solvers on the cost of SMT-based program analysis techniques. *Utopia* stores and reuses the solutions of formulas generated by these techniques to determine the satisfiability value of new formulas, without relying on possibly expensive calls to SMT solvers.

Differently from state-of-the-art approaches, which reuse solutions across formulas that can be shown to be equivalent to, contained in or implied by other formulas by exploiting a predefined sets of syntactical rules, *Utopia* reuses solutions across formulas that share at least one solution, regardless of their structural similarity, and thus strongly widens reuse opportunities.

The main contribution of *Utopia* is the introduction of *Sat-delta*, a heuristics that approximates the distance of a model from the solution space of a formula. *Sat-delta* enables the fast identification of formulas with similar solution spaces. In this way, *Utopia* quickly identifies a set of likely reusable solutions for a target formula and improves solution reusability.

The experimental results reported in the paper show that *Utopia* can identify on average more reuse opportunities and is considerably faster than competing approaches.

Acknowledgment

This work is partially supported by the Swiss SNF project CloSE (200021_149997/1) and by the Italian PRIN project IDEAS (2012E47TM2_006).

REFERENCES

[1] A. Aquino, F. A. 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*, ISSTA '15, pages 305–315. ACM, 2015.

[2] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in dynamic web applications. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '08, pages 261–272. ACM, 2008.

[3] E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proceedings of the International Conference on Software Engineering*, ICSE '13, pages 122–131. IEEE Computer Society, 2013.

[4] P. Braione, G. Denaro, and M. Pezzè. Enhancing symbolic execution with built-in term rewriting and constrained lazy initialization. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '13, pages 411–421. ACM, 2013.

[5] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the International Conference on Automated Software Engineering*, pages 443–446. IEEE Computer Society, 2008.

[6] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, OSDI '08, pages 209–224. USENIX Association, 2008.

[7] 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*, ICSE '11, pages 1066–1071. ACM, 2011.

[8] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS/ETAPS '08, pages 337–340. Springer, 2008.

[9] I. Erete and A. Orso. Optimizing constraint solving to better support symbolic execution. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, ICST '11, pages 310–315. IEEE Computer Society, 2011.

[10] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: Whitebox fuzzing for security testing. *ACM Queue*, 10(1):20–27, 2012.

[11] X. Jia, C. Ghezzi, and S. Ying. Enhancing reuse of constraint solutions to improve symbolic execution. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '15, pages 177–187. ACM, 2015.

[12] X. Li, Y. Liang, H. Qian, Y.-Q. Hu, L. Bu, Y. Yu, X. Chen, and X. Li. Symbolic execution of complex program driven by machine learning based constraint solving. In *Proceedings of the International Conference on Automated Software Engineering*, pages 554–559. ACM, 2016.

[13] T. Liu, M. Araújo, M. d'Amorim, and M. Taghdiri. A comparative study of incremental constraint solving approaches in symbolic execution. In *Proceedings of the Haifa Verification Conference*, HVC '14, pages 284–299. Springer, 2014.

[14] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*. IEEE Computer Society, 2008.

[15] J. Morse, L. Cordeiro, D. Nicole, and B. Fischer. Model checking ltl properties over ansi-c programs with bounded traces. *Software & Systems Modeling*, 14(1):65–81, 2015.

[16] H. Palikareva and C. Cadar. Multi-solver support in symbolic execution. In *Proceedings of the International Conference on Computer Aided Verification*, CAV '13, pages 53–68. Springer, 2013.

[17] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '08, pages 15–26. ACM, 2008.

[18] R. Sasnauskas, O. S. Dustmann, B. L. Kaminski, K. Wehrle, C. Weise, and S. Kowalewski. Scalable symbolic execution of distributed systems. In *Proceedings of the International Conference on Distributed Computing Systems*, ICDCS '11, pages 333–342. IEEE Computer Society, 2011.

[19] W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: Reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '12, pages 1–11. ACM, 2012.