Reusing Solutions Modulo Theories

Andrea Aquino, Giovanni Denaro, and Mauro Pezzè, *Senior Member, IEEE*

Abstract—In this paper we propose an approach for reusing formula solutions to reduce the impact of Satisfiability Modulo Theories (SMT) solvers on the scalability of symbolic program analysis. SMT solvers can efficiently handle huge expressions in relevant logic theories, but they still represent a main bottleneck to the scalability of symbolic analyses, like symbolic execution and symbolic model checking. Reusing proofs of formulas solved during former analysis sessions can reduce the amount of invocations of SMT solvers, thus mitigating the impact of SMT solvers on symbolic program analysis. Early approaches to reuse formula solutions exploit equivalence and inclusion relations among structurally similar formulas, and are strongly tighten to the specific target logics. In this paper, we present an original approach that reuses both satisfiability and unsatisfiability proofs shared among many formulas beyond only equivalent or related-by-implication formulas. Our approach straightforwardly generalises across multiple logics. It is based on the original concept of distance between formulas, which heuristically approximates the likelihood of formulas to share either satisfiability or unsatisfiability proofs. We show the efficiency and the generalisability of our approach, by instantiating the underlying distance function for formulas that belong to most popular logic theories handled by current SMT solvers, and confirm the effectiveness of the approach, by reporting experimental results on over nine millions formulas from five logic theories.

Index Terms—Symbolic program analysis, symbolic execution, SMT solver, solution reuse

1 Introduction

Symbolic program analysis determines the validity of program properties by reasoning on symbolic expressions. Program analyzers that rely on symbolic analysis, like symbolic executors [12] and symbolic model checkers [25], extensively use *Satisfiability Modulo Theories* (SMT) solvers to solve symbolic expressions and drive the analysis sessions.

State-of-the-art SMT solvers can efficiently handle huge expressions in some relevant logic theories, namely Booleans, Integers, Reals, the Mixed Theory of Integers and Reals, Strings, Fixed Size Bit-vectors, Arrays, Uninterpreted Functions and Uninterpreted Sorts [8], [9], [14], [15], [17], [19], and largely contribute to the industrial applicability of symbolic program analysis [5]. Despite the maturity of theories and tools, SMT solvers still represent a main bottleneck to the scalability of symbolic program analysis [24], [26], [28].

The intrinsic complexity of the problem of determining the satisfiability of a formula [16] indicates the need to investigate solutions to reduce the SMT bottleneck, beyond improving the intrinsic efficiency of the solvers themselves. Current research work moves along some promising research lines that include executing different solvers in parallel to mitigate the weaknesses of the individual solvers [26], augmenting SMT solvers with external techniques, like concrete and dynamic symbolic execution [18], and reducing the queries to SMT solvers by reusing solutions of formulas solved during the analysis session [2], [11], [20], [31].

In this paper, we move along the research line that has been early drafted within the preliminary caching frameworks that Cadar et al. introduced in *Klee* [11], refined in the seminal paper of Visser et al. who define *Green* [31], and further investigated with the *GreenTrie*, *Recal* and *Recal+* caching frameworks [2], [20]. The *Klee* cashing framework avoids repeated invocations of SMT solvers by simply searching for formulas that contain or are contained into formulas that are already solved in the ongoing analysis sessions. The *Green* framework normalises formulas to widen the reuse of formulas beyond the simple cases originally addressed in *Klee*. The *GreenTrie*, *Recal* and *Recal+* caching approaches define mature frameworks to handle inter-formula relationships beyond the simple equivalences and inclusions captured with *Klee* and *Green*.

All current frameworks work within the context of structurally similar formulas, and are strongly tighten to the specific target logics that each of them addresses. Thus they miss the opportunity of identifying reusable solutions shared among both structurally dissimilar formulas and formulas that are neither equivalent nor related by implication.

In this paper, we propose *Utopia*, an approach to efficiently identify formulas that likely share solutions independently from the structural similarities among formulas, and beyond formulas that are either equivalent or mutually contained. We introduce a concept of distance among formulas that approximates the similarities among solution spaces, and we provide efficiently computable instances of the distance function for the most popular logics that current SMT solvers address. We show that the *Utopia* distance does indeed widen the reusability of solutions across formulas, by comparing the

- *A. Aquino is with the Università della Svizzera italiana (USI), Lugano 6900, Switzerland. E-mail: andrea.aquino@usi.ch.*
- *G. Denaro is with the Università di Milano - Bicocca, Milano 20126, Italy. E-mail: giovanni.denaro@unimib.it.*
- *M. Pezzè is with the Università della Svizzera italiana (USI), Lugano 6900, Switzerland, and also with the Università di Milano - Bicocca, Milano 20126, Switzerland. E-mail: mauro.pezze@unimib.it.*

Manuscript received 8 Feb. 2018; revised 10 Dec. 2018; accepted 27 Jan. 2019. Date of publication 4 Apr. 2019; date of current version 14 May 2021. (Corresponding author: Giovanni Denaro.) Recommended for acceptance by V. Braberman. Digital Object Identifier no. 10.1109/TSE.2019.2898199

Utopia reuse rate with the reuse rate of the most promising competing approaches. We also show that the *Utopia* distance is generalisable across multiple logics by experimenting with the distance function instantiated for the most common logic theories that state-of-the-art SMT solvers address, namely Booleans, Integers, Reals, the Mixed Theory of Integers and Reals, Strings, Fixed Size Bit-vectors, Arrays, Uninterpreted Functions and Uninterpreted Sorts.

The main contribution of this paper is an original approach to identify a wide set of solutions reusable across formulas that may occur while symbolically analysing software programs. In details, this paper (i) proposes an original concept of distance between formulas that approximates the similarity of the solution spaces of both satisfiable and unsatisfiable formulas, (ii) implements the distance function for Booleans, Integers, Reals, Mixed Integers and Reals, Strings, Bit-vectors, Arrays, Uninterpreted Functions and Uninterpreted Sorts, (iii) presents a prototype implementation of the approach, and (iv) discusses the results of a set of experiments that show the improvement in reusing formulas in the context of symbolic program analysis.

We introduced a preliminary version of *Utopia* in [3], where we outline an approach to reuse solutions of only satisfiable formulas and only within the Quantifier-Free Integer and Real Arithmetic Logics. This paper presents (i) the general approach *Utopia* for a wide spectrum of logics, beyond the two only logics that we were able to address in our preliminary work, (ii) an original and general heuristic to efficiently reuse unsatisfiability proofs, while the ICSE paper discussed only the initial version of the heuristic that *Utopia* uses to address satisfiable formulas, and (iii) an extensive experimental evaluation of *Utopia* on over nine million formulas from five logic theories.

This paper is organised as follows. Section 2 recalls the terminology used in the paper, discusses the limitations of current approaches, and introduces the general solution that we propose in this paper. Section 3 defines *Sat-delta*, the heuristic distance between formulas that approximates the similarities of the solution spaces of the formulas, and instantiates *Sat-delta* for the theories of Booleans, Integers, Reals, Mixed Theory of Integers and Reals, Fixed Size Bit-Vectors, Strings, Arrays, and Uninterpreted Functions and Sorts. Section 4 defines *Unsat-footprint*, the heuristics that we propose to identify formulas that likely share unsatisfiability proofs. Section 5 presents the prototype implementation, and discusses in details a large set of experiments that answer the main research questions about effectiveness and efficiency of *Utopia*. Section 6 overviews the main characteristics of the related approaches to reuse formula solutions. Section 7 summarises the main contribution of this paper, and remarks the novelty with respect to the paper that we presented at the International Conference on Software Engineering in Buenos Aires in 2017 [3].

2 Reusing Solutions of Formulas

This paper introduces a caching framework to improve the scalability of symbolic program analysis techniques. The framework stores formulas and corresponding solutions computed during the analysis of a program, and then reuses the solutions to determine the satisfiability of new formulas met at future stages of the analysis of that program. Our caching framework overcomes the limitations of other recently proposed caching frameworks that only target specific logics and can reuse solutions only across specific classes of equivalent formulas or formulas related by implication. We first survey the characteristics and the main limitations of the state-of-the-art formula caching frameworks, and then illustrate the unique characteristics of the caching framework proposed in this paper.

2.1 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.

A *model* of a satisfiable formula is an assignment of values to the variables of a formula that makes the formula hold true. For instance, both assignments $\{[x = 11, y = 12]\}$ and $\{[x = 100, y = 200]\}$ are models of the formula $x > 10 \wedge y > x$.

An *unsatisfiable core* (in short *unsat-core*) of an unsatisfiable formula in conjunctive form is a subset of the conjuncts of that formula whose conjunction is itself unsatisfiable. For instance, there are three possible unsat-cores for the formula $x = 0 \wedge x \neq 0 \wedge x > 0$: $\{x = 0, x \neq 0\}$, $\{x = 0, x > 0\}$ and $\{x = 0, x \neq 0, x > 0\}$.

2.2 Structural Matching of Formulas

The caching frameworks proposed so far rely on matching the structure of two formulas to determine whether one formula is equivalent or related by implication to the other [2], [11], [20], [31]. These frameworks are all based on three main observations or a subset thereof: (i) two equivalent formulas share all solutions, (ii) a formula implied by a satisfiable formula shares all solutions with the latter, and (iii) a formula that implies an unsatisfiable formula is itself unsatisfiable. Based on these observations, these frameworks guarantee that if a formula matches another formula, then the solution of one formula can be reused to determine the satisfiability of the other one.

These frameworks have been applied mainly in the context of *symbolic execution* with promising results. However, their effectiveness strictly depends on the way in which they identify the equivalence or implication relations between two formulas. Currently, they exploit simple structural and mostly logic-dependent matching rules to determine whether a formula is equivalent to or implies another formula.

For instance, the frameworks that target formulas belonging to the *linear integer arithmetic* logic [2], [20], [31] include a rewrite rule that consists in normalising the linear inequalities in a formula from the form

$$c_1 x_1 + \cdots + c_n x_n < k,$$

to the form

$$c_1 x_1 + \cdots + c_n x_n \leq k - 1.$$

This rule is based on the observation that the expression $x < k$ has the same semantics of the expression $x \leq k - 1$, under the assumption that x is an integer value. While this assumption holds for the linear integer arithmetic, it does not hold for other logics, like the *linear real arithmetic*, where

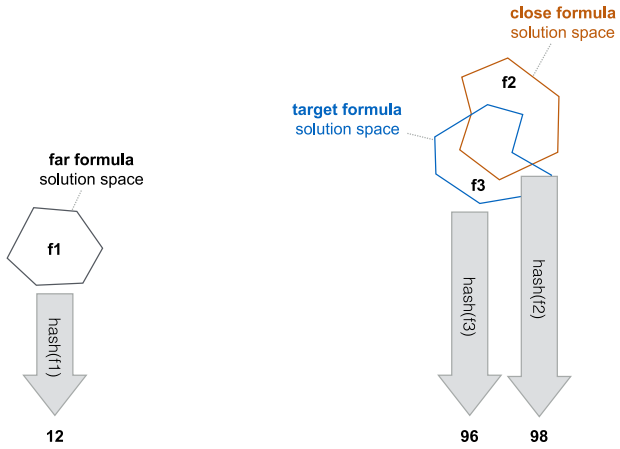

Fig. 1. *Utopia* polygon metaphor.

x is a real value. The mere presence of this rule tightens the applicability of this framework exclusively to the linear integer arithmetic logic.

Moreover, modern caching frameworks apply simple rules that cannot deduce the equivalence of many structurally different albeit equivalent formulas. For instance, the formulas

$$x \geq 0 \wedge y \geq 0 \wedge x + y = 10, \tag{1}$$

and

$$x = 5 \wedge y = 5, \tag{2}$$

belonging to the quantifier-free linear integer arithmetic logic, share exactly the same set of solutions (that is, only the model $\{[x = 5, y = 5]\}$), but no current approach can deduce their equivalence.

2.3 Beyond Structural Matching of Formulas

We build on the observation that the solution of a formula can be reused to determine the satisfiability or unsatisfiability of other formulas that share that solution, regardless of how different the structures or the solution spaces of the formulas are.

Let us consider for instance to have proved the satisfiability of formula

$$f : x + y \neq 0, \tag{3}$$

with model $[x = 1, y = 1]$, and to need to determine the satisfiability of formula

$$g : 2x - y < 3. \tag{4}$$

Current approaches cannot determine the satisfiability of g from f, since the two formulas are neither equivalent nor related by implication. However, the model $[x = 1, y = 1]$ built to solve f satisfies also g, thus we could deduce that g is satisfiable by referring to the model derived for f.

Likewise, let us consider to have proved the unsatisfiability of formula

$$f : x > 1 \wedge x \neq 9 \wedge x < 0,$$

with the unsat-core $\{x < 0, x > 1\}$, and to need to determine the satisfiability of formula

$$g : x < 0 \wedge x > 1 \wedge x + k = 0 \wedge k = 1.$$

Current approaches cannot determine the unsatisfiability of g from f, since the two formulas are neither equivalent nor related by implication. For example, the approach [11] embodied in the symbolic executor *Klee*, would check whether all clauses of f are included in g, which is not the case. However, the unsat-core of f is contained in g, thus we could deduce that g is unsatisfiable by referring to the unsat-core derived for f.

Our approach, *Utopia*, is based on the intuition that it is possible to determine the satisfiability of a target formula by *directly* reusing the solution of another formula that shares some solutions with the target formula, and has been already solved in the past. *Utopia* works by selecting a set of candidate solutions from the available ones, and directly testing these solutions on the target formula. *Utopia* simply tests the solution of a solved formula on a not-yet-solved one, without requiring any specific relation between the structures of the involved formulas.[1]

Testing solutions among millions of candidates may quickly turn to be more expensive than solving the formula directly with an SMT solver, however, testing only few candidate solutions is rather inexpensive, and can be an excellent way for reusing solutions across formulas. *Utopia* heuristically selects a small set of candidate solutions from a large set of solutions of formulas solved in the past, by introducing a *distance function* that approximates the relation between formulas that share solutions.

Ideally, the *Utopia distance function* d between two formulas estimates the likelihood that a solution of the first formula is shared by the second formula:

$$d : F \times F \rightarrow \mathbb{R},$$

where F denotes a set of formulas. Thus, *Utopia* can use the distance d to rank already solved formulas according to the likelihood that they share a solution with a target formula f, and identify a small set of candidate solutions to test against f: the solutions of the formulas closest to f.

Utopia exploits d by defining an approximation $\tilde{d}$ that works based on hash-codes computed on the formulas. The approximation $\tilde{d}$ allows *Utopia* to effectively identify the formulas closest to a target one by simply comparing the hash-codes of the formulas, without requiring to directly compare all pairs of formulas.

Intuitively, *Utopia* uses the easily-computable approximation $\tilde{d}$ to position formulas in a numeric space, and refers to such simplified space to identify the formulas whose solutions are the closest to the target formula. Fig. 1 illustrates the *Utopia* intuition with the metaphor of distances of polygons that represent the solution spaces of formulas. In the figure, the solution spaces of formulas $f2$ and $f3$ are

1. *Utopia* pairs the variables in the formulas with the variables in the solutions according to the positional mapping induced by the order in which the variables appear in the formulas and in the solutions, independently from the variable names. Thus, *Utopia* is independent from variable names. *Utopia* reuses a solution for formulas with less variables than the solution by dismissing the additional variables in the solution, and for formulas with more variables than the solution by using default values, for instance the constant '0' for integer variables.

mutually close, while the solution space of formula $f1$ is far away. Thus, the distances among the solution spaces suggest that a previously computed solution for formula $f2$ can be a good candidate for solving formula $f3$, while a previously computed solution for formula $f1$ is not. As illustrated in the figure, *Utopia* works by mapping formulas $f2$ and $f3$ to hash-codes close each-other, and $f1$ to a far-away hash-code. Therefore, to prove $f3$ after proving formulas $f1$ and $f2$, *Utopia* would compute the hash-code of $f3$, identify $f2$ as the closed already-proved formula, and try to reuse the solution of $f2$ for $f3$.

Formally, *Utopia* computes the distance $\tilde{d}$ with respect to a hash-function h that represents a suitable mapping from formulas to elements of a numerical set H. For two formulas f and g, *Utopia* relies on the heuristic that

$$d(f,g) \approx \tilde{d}(h(f),h(g)),$$

where

$$h : F \to H \text{ and } \tilde{d} : H \times H \to \mathbb{R}.$$

Based on this approach, *Utopia* computes the h-value of each formula only once, and efficiently identifies the formulas that are more likely to share solutions with a new formula f by choosing the solutions of the previously-solved formulas with h-value closest to the h-value of f, and can do it efficiently by relying on $\tilde{d}$.

In Section 3 we present *Sat-delta*, a concrete implementation of the hash-function h that numerically captures the behaviour of a satisfiable formula by evaluating the formula with respect to a set of pre-defined reference models. We then define the distance function $\tilde{d}$ as the absolute difference of the *Sat-delta* values of two formulas. In this setting, *Utopia* keeps the stored formulas ordered by their *Sat-delta* values and can thus efficiently retrieve the stored formulas that are the closest to a given target formula by means of a k-nearest-neighbours search. Similarly, in Section 4 we present *Unsat-footprint*, a concrete implementation of h that captures the behaviour of an unsatisfiable formula with a hash-code based on Bloom filters [4]. In this case, the distance function $\tilde{d}$ corresponds to the bitwise-and of the *Unsat-footprint* values of associated with the formulas, which yields zero for the unsat-cores that could be shared with a given target formula, and is different than zero otherwise.

Utopia can cope with many logics, provided it is instantiated with hash functions h that do not rely on any logic-specific transformations of the formulas.

2.4 *Utopia* Algorithm for Reusing Solutions

Algorithm 1 presents the *Utopia* algorithm that efficiently retrieves a reusable solution for a formula, given:

- the formula f to be solved;
- two repositories S and U of already solved formulas paired with the corresponding solutions, where S contains satisfiable formulas each paired with a corresponding solution, being the solutions represented as formula models, and U contains unsatisfiable formulas each paired with a corresponding solution, being the solutions represented as unsat-cores of the formulas, respectively;
- two distance functions $\tilde{d}_S$ and $\tilde{d}_U$, and two corresponding hash-functions h_S and h_U, respectively, which instantiate the technique described in Section 2.3 for the efficient selection of candidate solutions out of each repository, where $\tilde{d}_S$ and h_S are specialised to estimate the (solution sharing) distance between satisfiable formulas, and $\tilde{d}_U$ and h_U for the distance between unsatisfiable formulas, respectively;
- two positive integers n_S and n_U that bound the (small) amount of candidate solutions that the algorithm is allowed to select out of each repository;
- a backend $SOLVER$ that the algorithm uses to compute the solution of the formula f when it fails to find any reusable solution.

We present concrete instances of the distance functions $\tilde{d}_S$ and $\tilde{d}_U$, along with the corresponding hash-functions h_S and h_U, in the next sections.

Algorithm 1. The *Utopia* Algorithm

Require:
f the target formula to be solved.
S a repository of ⟨ formula, model ⟩ pairs.
U a repository of ⟨ formula, unsat-core ⟩ pairs.
$\tilde{\mathbf{d}}_\mathbf{S}$ the distance function for satisfiable formulas.
$\tilde{\mathbf{d}}_\mathbf{U}$ the distance function for unsatisfiable formulas.
$\mathbf{h_S}$ the hash-function for the computation of $\tilde{\mathbf{d}}_\mathbf{S}$.
$\mathbf{h_U}$ the hash-function for the computation of $\tilde{\mathbf{d}}_\mathbf{U}$.
$\mathbf{n_S}$ the maximum amount of candidate models.
$\mathbf{n_U}$ the maximum amount of candidate unsat-cores.
SOLVER the SMT solver.

1: $f_S = h_S(f)$
2: $models \leftarrow \text{CLOSEST}(S, \tilde{d}_S, f_S, n_S)$
3: **for** $m \in models$ **do**
4: **if** isSolution(f, m) **then**
5: **return** m ▷ m is a reusable solution
6: **end if**
7: **end for**
8: $f_U = h_U(f)$
9: $unsat-cores \leftarrow \text{CLOSEST}(\text{U}, \tilde{d}_U, f_U, n_U)$
10: **for** $u \in unsat-cores$ **do**
11: **if** isSolution(f, u) **then**
12: **return** u ▷ u is a reusable solution
13: **end if**
14: **end for**
15: $solution \leftarrow \text{SOLVER}(f)$
16: **if** isModel$(solution)$ **then**
17: $\text{STORE}(S, f, solution, f_S)$
18: **return** $solution$
19: **else if** isUnsatCore$(solution)$ **then**
20: $\text{STORE}(U, f, solution, f_U)$
21: **return** $solution$
22: **else**
23: **return** None ▷ The solver did not yield solutions.
24: **end if**

Given a formula *f*, *Utopia* identifies either a model satisfying *f* or an unsat-core proving that *f* is unsatisfiable, by searching two repositories that contain satisfiable formulas paired with their models (lines 1–7) and unsatisfiable formulas paired with their unsat-cores (lines 8–14), respectively. In detail, *Utopia* selects a small set of candidate solutions for a target formula by computing the values of h_S and h_U for the

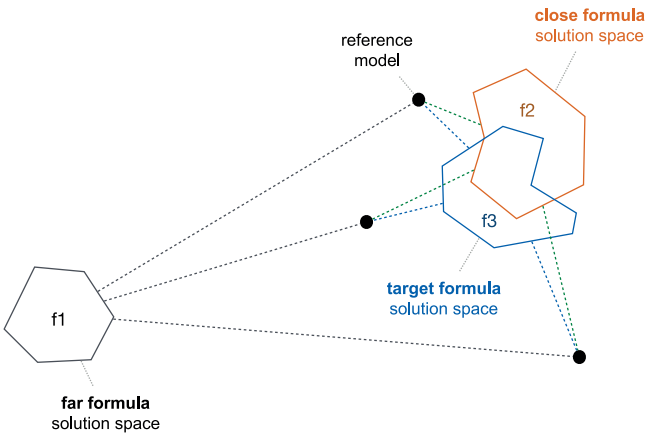

Fig. 2. Metaphor of how *Utopia* uses the *Sat-delta* heuristic.

target formula (lines 1 and 8, respectively), and retrieving the set of n_S and n_U formulas with the h_S and h_U values closest to the ones of the target formula, according to the distance functions $\tilde{d}_S$ and $\tilde{d}_U$, respectively (lines 2 and 9). Finally, *Utopia* checks if any of the identified solutions can be reused to determine the satisfiability of the target formula (with the loops starting at lines 3 and 10, respectively). If this is not the case, *Utopia* relies on an external SMT solver to solve the target formula (line 15) and updates the relevant repository with the generated solution (lines 17 and 20).

Utopia is parametric with respect to n_S and n_U, the numbers of candidate solutions retrieved from each repository. As discussed in Section 5, in the experiments reported in this paper we configured *Utopia* setting both n_S and n_U to 10. In this configuration, *Utopia* selects and tries to reuse 10 models and 10 unsat-cores to solve a target formula.

Utopia may be subject to both false-negatives and false-positives. A false-negative is an available solution-sharing formula whose hash-code is farther from f than other formulas in the corresponding repository, leading *Utopia* to miss this solution. The false-positives are solutions that *Utopia* first selects as candidate and then verifies that are not solutions. Both false-positives and false-negatives are primarily due to the intrinsically heuristic nature of the h_S and h_U approximations. Some false positives may depend on formulas that share many solutions with the target formula, but not the currently available one. *Utopia* safely handles false-positives by design, since it discards them by directly evaluating the target formula on the candidate solutions. The experiments that we discuss later in the paper indicate that false-negatives have a negligible impact.

3 The Sat-Delta Heuristic

Algorithm 1 requires efficient instantiations of the general hash and distance functions (h_S and $\tilde{d}_S$, respectively), to efficiently retrieve a set of solutions likely reusable for the target formula. *Utopia* instantiates the general hash-function h_S with *Sat-delta*, a heuristic hash-function that maps formulas on numbers by (i) evaluating the variables in the formulas with respect to a common set of reference models, and (ii) exploiting the structure of the formula to characterise the extent by which the reference models miss the solution space of the formula. The underlying intuition is that, by characterising the formulas with respect to a common set of reference models, *Utopia* increases the chances of assigning similar values to formulas that share some models. Thus *Utopia* computes $\tilde{d}_S$ as the difference of *Sat-delta* values. The experimental results reported in Section 5 confirm that *Sat-delta* is indeed an efficient concretization of h_S.

Fig. 2 illustrates *Sat-delta* with the polygon metaphor introduced In Fig. 1. The black dots represent the reference models, and the distance of the polygons from three reference points (the dashed lines from the black points) represent *Sat-delta*. As polygons with similar distances from the reference points have higher chance to overlap, formulas with similar distances from the reference models have higher chance to have closer solution spaces, and thus solutions reusable across them.

Given a *target formula* $f3$ the satisfiability of which has to be determined (the blue polygon in the figure), *Sat-delta* identifies the models of the closest formula $f2$ (the orange polygon in the figure) as the most likely reusable models for the target formula: those models might indeed belong to the intersection of the two polygons in the Euclidian space.

We define the *Sat-delta* value of a formula f with respect to a reference model m as the least change to transform model m into a model that satisfies f. For instance, the *Sat-delta* value of formula $x \leq 1$ with respect to the model $[x = 3]$ is 2, since the least change of the model $[x = 3]$ to obtain a model $[x = 1]$ that satisfies the formula is to subtract 2 from the model

$$Sat - delta(x \leq 1 \text{ w.r.t. } [x = 3]) = 2.^{2}$$

We generalise the *Sat-delta* value of a formula f with respect to a set of reference models $\{m_1, \ldots, m_n\}$ as the average of the *Sat-delta* values of f with respect to each model m_i in the set.

Utopia exploits the heuristically-grounded hypothesis that the similarity of the *Sat-delta* values of formulas is a proxy of the similarity of the behaviour of the formulas with respect to a set of reference models, which in turn is a proxy of the similarity of the solution spaces of the two formulas. Formulas with similar distances from the reference models have higher chance to have closer (and thus more likely overlapping) solution spaces than formulas with very different distances from the same reference models. Because of this, given a *target formula* the satisfiability of which has to be determined (for example, formula $f3$ in Fig. 2), *Sat-delta* suggests that it is more convenient to try to reuse first the models that are available for *close formulas* (for example, formula $f2$ that shares many solutions with formula $f3$) rather than the ones that are available for *far formulas* (for example, formula $f1$ that share no solution with formula $f3$).

In this section, we define a *Sat-delta* heuristic that targets formulas belonging to many popular quantifier-free logics. Our definition of *Sat-delta* applies to most of the quantifier-free logics defined by the SMT-LIB standard, which is the reference standard supported by many state-of-the-art SMT

2. In this simple example, the value of *Sat-delta* induces a satisfying model of the formula (as a change of the reference model). This is not the general case, since *Sat-delta* works by arithmetically aggregating the *Sat-delta* values of the atomic formulas across the boolean structure of the formula. Section 3.1 resents the complete algorithm.

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

Fig. 3. *Sat-delta* of atomic formulas in the theory of integers.

solvers. Formulas belonging to quantifier-free SMT-LIB standard logics are propositional compositions of atomic formulas over expressions in different reference theories, such as the theory of integers and the theory of strings; Atomic formulas are boolean function relating expressions in the theory, and expressions are combinations of functions, variables and constants defined in the theory.

For instance, the formula $x \leq 0 \wedge y + 1 > x$, where x and y denote two integer variables, belongs to the quantifier-free linear integer arithmetic logic, a logic that allows reasoning on quantifier-free formulas over linear inequalities over integer variables and constants. The formula is the logical conjunction of the two atomic formulas $x \leq 0$ and $y + 1 > x$, where each atomic formula is a linear inequality over integer expressions.

Similarly, the formula

$$\text{prefix} - \text{of}(''\text{hello}'', \text{s}) \vee \text{suffix} - \text{of}(''\text{world}'', \text{s}),$$

belongs to the theory of strings, and is the logical disjunction of two atomic formulas, where each atomic formula is a boolean function ($\text{prefix} - \text{of}$ and $\text{suffix} - \text{of}$), evaluated with respect to string expressions.

Sat-delta targets quantifier-free logics over atomic formulas in any of the following reference theories:

- booleans, also known as the core theory,
- integers,
- reals,
- mixed integers and reals,
- fixed size bit-vectors,
- strings,
- arrays,
- uninterpreted functions and sorts.

In the next sections, we first recall the definition of *Sat-delta* for the theory of integers, by referring to the computation algorithm that we presented in [3]. Then we properly extend the definition of *Sat-delta* to the other theories, by discussing the details related to the construction of the reference models and the rules to compute *Sat-delta* with respect to the other theories. Finally, we discuss how *Utopia* exploits *Sat-delta* to efficiently retrieve reusable models for new formulas by searching in a large repository containing formulas solved in the past paired with their models.

3.1 *Sat-Delta* for the Theory of Integers

We define *Sat-delta* by specifying (i) the reference models used to compute *Sat-delta*, (ii) the rules for computing the *Sat-delta* value of the atomic formulas in the theory, and (iii) the rules for computing the *Sat-delta* value of the propositional composition of several atomic formulas in the theory.

3.1.1 Reference Models

In the theory of integers, a reference model is a concrete assignment of integer values to the variables of the considered formulas. To compute *Sat-delta* on arbitrary formulas, we need reference models that are *total* with respect to the variables that may occur in the formulas, that is, a model that assigns a concrete integer value to each variable that may appear in any formula, so that, when a formula is evaluated on such model, it is always possible to decide whether or not the model satisfies the formula. For instance, the model $[x = 0]$, which assigns value 0 to all occurrences of variable x in a formula is not total, since it does not enable to evaluate formula $x + y > 0$, because it does not assign any value to variable y. Conversely, a model that assigns value 0 to all integer variables in a formula, which hereon we denote as $[\forall v \,|\, m(v) = 0]$, is total with respect to all formulas defined in the theory of integers.

All models of the form $[\forall v \,|\, m(v) = k]$, where $k \in \mathbb{Z}$ is any integer constant, are total in the theory of integers, and can be used as reference models for calculating *Sat-delta*.

The choice of the reference models can impact the precision of the approximation of *Sat-delta*. In our current prototype of *Utopia* (Section 5.3) we experimented *Sat-delta* with different reference models, and confirmed the intuition that in general computing *Sat-delta* by averaging the distance from a set of reference models approximates the distance of the solution spaces more accurately than computing *Sat-delta* with respect to a single reference model. Intuitively, the distance from a single model cannot distinguish the different relation between small and large solution spaces (small and large polygons in Fig. 2), while referring to a set of models can better capture the relation between spaces of different size. Our experiments also indicate that oddly distributed reference models better approximate the distance among solution spaces that evenly distributed ones, and that a small set of reference models already suffices to improve the precision of the approximation, while the additional improvement that derives from considering large sets of reference models is often negligible.

In the experiments reported in Section 5, we compute *Sat-delta* for formulas in the theory of integers as the average of the *Sat-delta* values with respect to the following three oddly distributed models:

$[\forall v \,|\, m(v) = 0]$
$[\forall v \,|\, m(v) = 100]$
$[\forall v \,|\, m(v) = -1000]$

3.1.2 Sat-Delta of Atomic Formulas

In the theory of integers, atomic formulas are either equations or inequalities over integer expressions, that is, formulas in the form $e_1 \odot e_2$, where e_1 and e_2 are arithmetic expressions over integer variables and constants, and $\odot$ is a comparison operator in the set $\{=, \neq, <, \leq, >, \geq\}$. For instance, the formula $x + y > 0$ is an atomic formula consisting of a single linear integer inequality.

We define the *Sat-delta* value of an inequality with respect to a total reference model as the smallest positive integer value that must be added to either side of the inequality to make the model satisfy it. As shown in Fig. 3, the *Sat-delta* value of an atomic formula is 0 if the model satisfies the formula, a positive value otherwise, consistently with the polygon metaphor illustrated in Fig. 2. For instance, the *Sat-delta* value of the inequality $2x + 3 < y$ with respect to the model

TABLE 1
Sorts and Atomic Formula Operators by Theory

Theory	Sorts	Operators
Booleans	$Bool$	C
Integers	Int	C ∪ A
Reals	$Real$	C ∪ A
Mixed Integer/Reals	$Int, Real$	C ∪ A ∪ {is − int}
Strings[3]	$String$	$C \cup S$
Fixed Size Bit-vectors	$BitVec_n$	C ∪ (A × {s, u})
Uninterpreted Sorts	$USort_s$	C
Arrays	$Array[I \rightarrow V]$	∅
Uninterpreted Functions	$\lambda P_1, \ldots, P_n \rightarrow V$	∅

In the table above,
$C = \{=, \neq\}$,
$A = \{>, \geq, \leq, <\}$, *and*
$S = \{\text{match}, \text{contains}, \text{prefix} - \text{of}, \text{suffix} - \text{of}\}$.

$[\forall v \,|\, m(v) = 0]$ is 4, since 4 is the smallest positive integer that when added to the formula, in this case to the right side of the inequality, produces a new inequality, in this case $2x + 3 < y + 4$, which is satisfied by the model.

3.1.3 Sat-Delta of Formulas

We compute the *Sat-delta* value of general formulas with respect to a total reference model, by aggregating the *Sat-delta* values of the atomic formulas contained in the formula, thus exploiting the propositional structure of the formulas. We discuss the algorithm for the functionally complete set of connectives of the logical operators $\neg$ (negation), $\wedge$ (conjunction) and $\vee$ (disjunction).

The algorithm for computing *Sat-delta* for a formula f works as follows:

i) It transforms the formula f into an equivalent formula that does not contain negation operators, by removing the negations from f;
ii) it computes the *Sat-delta* values of the atomic formulas that compose the obtained formula, and substitutes the the atomic formulas with the computed *Sat-delta* values;
iii) It transforms the formula into an arithmetic expression by mapping conjunctions and disjunctions in the formula on sum and minimum operations, respectively;
iv) It computes the *Sat-delta* value of the original formula as the result of the obtained arithmetic expression.

The algorithm removes the negations in the formula (Step (i)) by safely pushing the negation operators down in the formula and transforming the negated atomic formulas into equivalent non-negated one. It pushes the negation operators down in the formula by repeatedly applying the De Morgan's laws, until the only negated sub-formulas are atomic formulas. It transforms the negated atomic formulas into equivalent non-negated formulas by applying elementary equivalences among formulas. For instance, the algorithms transforms formula $\neg(x \leq 0 \vee y > 1)$ the equivalent formula $\neg(x \leq 0) \wedge \neg(y > 1)$ by applying De Morgan's law, and then into the equivalent formula $x > 0 \wedge y \leq 1$, which does not contain negation operators.

3. We refer to the theory of strings proposed by Liang [23].

The algorithm maps the formula into an arithmetic expression over the *Sat-delta* values corresponding to the atomic formulas contained in the original formula, by replacing conjunctions of atomic formulas with the sum of the *Sat-delta* values of the atomic formulas, and disjunctions of sub-formulas with the minimum of the *Sat-delta* values the atomic formulas. The mapping follows the intuition that satisfying conjunctions requires satisfying all conjuncts, and thus every non-satisfied conjunct contributes a missed amount, that is, its *Sat-delta* value, to the total by which the given reference model does not satisfy the overall conjunction, while satisfying a disjunction requires satisfying either of the disjuncts, and thus it would suffice to satisfy the least missed one, that is, the one with minimum *Sat-delta* value. For instance, the algorithm computes the *Sat-delta* value of the formula $(x > 0 \vee y > 1) \wedge \neg(x + y \leq 2)$ with respect to the reference model $[\forall v \,|\, m(v) = 0]$, by

i) inverting the last atomic formula to remove the corresponding negation, obtaining the formula $(x > 0 \vee y > 1) \wedge x + y > 2$,
ii) computing the *Sat-delta* values of the atomic formulas with respect to the reference model:
- *Sat-delta* $(x > 0$ w.r.t. $[\forall v \,|\, m(v) = 0]) = 1$
- *Sat-delta* $(y > 1$ w.r.t. $[\forall v \,|\, m(v) = 0]) = 2$
- *Sat-delta* $(x + y > 2$ w.r.t. $[\forall v \,|\, m(v) = 0]) = 3$

iii) replacing the atomic formulas with their corresponding *Sat-delta* values, and the logical operators $\wedge$ and $\vee$ with the arithmetic operators $+$ and $\min$, respectively, obtaining the arithmetic expression $\min(1, 2) + 3$ that evaluates to 4.

In summary, given the model $[\forall v \,|\, m(v) = 0]$, it holds that

$$\begin{aligned} &Sat - delta((x > 0 \vee y > 1) \wedge \neg(x + y \leq 2)) \\ &= Sat - delta((x > 0 \vee y > 1) \wedge x + y > 2) \\ &= min(1, 2) + 3 \\ &= 4. \end{aligned}$$

As discussed earlier in this section, we define the *Sat-delta* value of a formula with respect to a set of total reference models as the average of the *Sat-delta* values computed with respect to each reference model in the set.

The algorithm to compute *Sat-delta* for propositional formulas can be easily generalised to other theories by defining *Sat-delta* for the atomic formulas of the theory, and introducing appropriate rules to remove negations in such theories. In the next section we describe a generalisation of *Sat-delta* to cope with most standard theories.

3.2 Generalised *Sat-Delta*

We generalise *Sat-delta* to the theories in Table 1: Column *Theory* indicates the name of the theory; Column *Sort* indicates the sorts of the variables in each theory; Column *Operators* indicates the operators that can be used to build atomic formulas in each theory. We extend the definition of *Sat-delta* to these theories by defining (i) the total reference models that we use to assign concrete values to the variables in these theories, and (ii) the rules for computing the *Sat-delta* values of the atomic formulas that can be defined in these theories.

For example, the second row of the table indicates that the definition of *Sat-delta* for the formulas in the theory of

integers shall include both total reference models that assign constants to integer variables, and computation rules for atomic formulas built with the comparison operators $\{=, \neq, >, \geq, <, \leq\}$, consistently with the definition that we discussed in the previous section.

Below, we present the reference models and the rules to compute *Sat-delta* for formulas in the theories listed in Table 1. We recall that the computation rules that aggregate the *Sat-delta* values of the atomic formulas across the propositional structure of the formulas, presented in the previous section with reference to the theory of integers, are the same for all theories.

3.2.1 Reference Models

For each theory, the reference models map variables of the sorts that characterise the theory to constant values. We define the mapping of variables to concrete values first for *primitive* and then for *structured* sorts. Primitive sorts are sorts for which we can identify a concrete constant that can be assigned to all variables of that sort; Structured sorts correspond to parametric and recursive data types.

With reference to column *Sorts* in Table 1, primitive sorts include the sorts Bool, Int, Real and String. The simple mapping of integer variables to integer constants that we defined in the former section can be straightforward generalized to all primitive sorts: for the theory of integers, we built reference models that cover all variables of Int sort by simply mapping all variables of such sort to an integer constant $k \in \mathbb{Z}$ chosen as a concrete representative value. In line with this approach, we build reference models for sort Bool by mapping all variables of boolean sort to either the constant *true* or *false*, for sort Real by mapping all variables of real sort to a chosen real number, and for sort String by mapping all variables of string sort to a chosen string constant.[4]

Structured sorts can be instantiated in infinitely many ways by varying the values of either the parameters or the inner sorts that they depend on, and thus they represent families of primitive sorts. The structured sorts in the table are fixed size bit-vectors (sort $BicVec_n$), Arrays (sort $Array[I \rightarrow V]$) and Uninterpreted Functions (sort $\lambda P_1, \ldots, P_n \rightarrow V$). Sort $BicVec_n$ is parametric with respect to the size (the number of bits) of the bit-vector it describes. Thus, a formula in the theory of fixed size bit-vectors can include expressions that refer to bit-vectors of different size, such as bit-vectors of eight, sixteen and sixty-four bits. Sort $Array[I \rightarrow V]$ is parametric with respect to the types of both the indexes and the elements of the array. Thus a formula in the theory of arrays can include expressions that refer to arrays defined over different types, such as arrays with both indexes and elements of integer type, and arrays with indexes of string type and elements of boolean type. Sort $\lambda P_1, \ldots, P_n \rightarrow V$ is parametric with respect to the types specified in the signature of the function it describes. Thus a formula in the theory of uninterpreted functions can include expressions that refer to functions with different signatures.

For structured sorts there exists no single constant that can be assigned to all variables of the sort, since distinct variables can be instantiated with respect to different parameters, and thus require different constants. For example, a formula in the theory of fixed size bit-vectors can include variables that represent bit-vectors of distinct sizes, and thus shall be assigned with constant bit-vectors of appropriate size each. Thus, we build reference models for structured sorts by defining parametric functions that return constants based on the parameters of the sort.

We define total reference models for fixed size bit-vectors as functions that return a constant bit-vector for each possible bit-vector size, thus providing a compatible literal for any bit-vector variable of any possible size. For example, a total reference model may assign any bit-vector variable of size i to the constant bit-vector obtained by extracting the first i bits of an arbitrarily chosen infinite sequence of zeros and ones. As a concrete case, by choosing an infinite sequence of zeros, we build a total reference model that assigns all bit-vector variables to a bit-vector of the appropriate size representing the value zero.

The sorts of arrays and uninterpreted functions are recursively defined by means of sorts they take as parameters: The array sort defines a data structure that maps indexes of sort I to values of sort V; The theory of uninterpreted functions is defined with respect to the function term, a mathematical function that maps parameters of sorts P_1, ..., P_n to values of sort V. Formulas in this theory include expressions describing functions and their properties. For instance, the formula $f(5) = 0$, where f is a function with integer domain and range, asserts that there exists some function that maps the integer value 5 to 0.

We define total reference models for the recursive sorts as functions that assign the variables of the sorts to compatible constants, building these constants on top of the reference models that correspond to the (sort) parameters referred in the variables. In particular, for the array and the uninterpreted function sorts, we specify reference models that associate (i) each array variable with a concrete array that maps all indexes to a constant value of the sort of the values contained in the array, and (ii) each uninterpreted function variable with a concrete function that always returns a constant value of the sort of the values returned by that function, respectively.

Formally, we assign all array variables of sort $Array[I \rightarrow V]$ to the constant array that maps any index of sort I to a (possibly recursively built) reference model of sort V, and we assign all uninterpreted function variables of sort $\lambda(P_1, \ldots, P_n) \rightarrow V$ to the constant function that maps all inputs of the appropriate sorts $P_1, \ldots, P_n$ to a reference model of sort V. For instance, to define a reference model that assigns all variables of sort $Array[String \rightarrow Int]$, we refer to the reference model $[\forall v \,|\, m(v) = k]$ defined for variables of sort Int, to build the constant associative array that maps any (string) index to the constant k. Similarly, to define a reference model that assigns all uninterpreted function variables of sort $\lambda(Real) \rightarrow Int$, we build the constant function that returns the constant k for any input.

The theory of uninterpreted functions also introduces the concept of uninterpreted sorts, which represent sets of uninterpreted objects identified purely by their name. For instance, the sort A and the sort B denote sets of uninterpreted objects of type A and B, respectively. Uninterpreted

4. The reference models for the mixed theory of integers and reals, include both a reference model for sort Int and a reference model for sort Real.

TABLE 2
Rules to Remove Negations

Negated atomic formulas	Corresponding inverted formulas
$\neg(e_1 = e_2)$	$e_1 \neq e_2$
$\neg(e_1 \neq e_2)$	$e_1 = e_2$
$\neg(e_1 \text{contains} e_2)$	$\text{first} - \text{index}(e_2, e_1) < 0$
$\neg(e_1 \text{prefix} - \text{of} e_2)$	$\text{first} - \text{index}(e_1, e_2) \neq 0$
$\neg(e_1 \text{suffix} - \text{of} e_2)$	$\text{last} - \text{index}(e_1, e_2) \neq \text{len}(e_2) - \text{len}(e_1)$
$\neg(e \text{matchr} e)$	$\text{ite}(e \text{matchr} e, e \neq e, e = e)$
$\neg(\text{is} - \text{int}(e))$	$e - \text{to} - \text{real}(\text{to} - \text{int}(e)) \neq 0$

sorts support formulas that predicate on the equality or distinctness of the values of some uninterpreted sort returned by some functions. For these formulas, we assign all uninterpreted functions returning values of a given uninterpreted sort to the constant function that always returns an arbitrarily chosen constant object of that type.

3.2.2 Computing Sat-Delta

The algorithm presented in Section 3.1 computes the *Sat-delta* value of generic formulas expressed as propositional combinations of multiple atomic formulas, starting from the *Sat-delta* values of the atomic formulas. In this section, we define *Sat-delta* of the atomic formulas that can be expressed in the theories listed in Table 1, by referring to the operators reported in the table. We define *Sat-delta* for the new kinds of formulas following the core principle that we introduced in Section 3.1 for the theory of integers: The *Sat-delta* value of a formula f with respect to a reference model m is the minimal amount of modifications of the model m that produce a model that satisfies f.

Below, we present the formal definitions of *Sat-delta* for all the operators in Table 1, that is, the inequalities (operators $=, \neq, >, \geq, <, \leq$), which are shared among most theories (Booleans, Integers, Reals, Mixed Integer/Reals, Fixed Size Bit-Vectors, Unint. Sorts), the operators specific for the String theory, and the operator *is-int* of the Mixed Integer/Reals theory.

Inequalities. In the theories of Booleans, Integers and Reals, inequality atomic formulas are expressed over arithmetic expressions denoting booleans, integer numbers and real numbers, respectively. In the theory of Fixed Size Bit-Vectors, inequalities are expressed over expressions denoting bit-vectors of the same size. The Fixed Size Bit-Vectors expressions can be interpreted as either signed or unsigned numbers, as explicitly indicated with subscripts s (signed) and u (unsigned) of the operators: $<_s, <_u, \leq_s, \leq_u, >_s, >_u, \geq_s, \geq_u$.

We define *Sat-delta* for inequality atomic formulas by extending the definition in Fig. 3: The *Sat-delta* value of an inequality atomic formula with respect to a reference model is the smallest non-negative integer value that must be added to either side of the formula to make the model satisfy it. In the theory of Fixed Size Bit-Vectors, we compute *Sat-delta* consistently with the signed or unsigned interpretation of the values, according to the comparison operator used in each specific formula.

Operators for Strings. Atomic formulas in the theory of strings are expressions either of the form $s \odot t$, where s and t are expressions denoting strings and $\odot \in \{=, \neq, \text{contains}, \text{prefix} - \text{of}, \text{suffix} - \text{of}\}$, or of the form $s \text{matchr} e$, where s is an expression denoting a string, and re is a regular expression. Expressions $s \odot t$ assert that the string s equals, is different from, contains, is a prefix of or is a suffix of the string t, respectively. Expressions $s \text{matchr} e$ asserts that the string s is part of the regular language defined with the regular expression re.

We compute the *Sat-delta* value of $s \odot t$ atomic formulas with respect to a reference model m, by evaluating the expressions s and t on the model m to obtain two constant strings $m(s)$ and $m(t)$. We compute the *Sat-delta* value of a formula of the form $s = t$ by calculating the minimum number of characters that must be added, removed or modified in the string $m(s)$ to obtain the string $m(t)$. This measure is commonly known as the *edit distance* of the strings $m(s)$ and $m(t)$ [22]. For example, the *Sat-delta* value of the formula $x \text{concat} y = goodafternoon$ with respect to the model $m = \{x = good, y = noon\}$ is the edit distance dist(goodafternoon, goodnoon) = 5, since we need to add 5 letters to goodnoon to obtain goodafternoon.

We compute the *Sat-delta* value of $s \neq t$ formulas with respect to a model m by comparing the concrete strings $m(s)$ and $m(t)$ obtained instantiating s and t on m: *Sat-delta* =0 if $m(s) \neq m(t)$, 1 otherwise, since changing a character in either strings turns equal strings into different ones.

We compute the *Sat-delta* value of a formula of the form $s \text{contains} t$ with respect to a model m as the edit distance between $m(t)$ and $m(s)$, if $m(s)$ is shorter than $m(t)$, or the minimum edit distance between $m(t)$ and any substring of $m(s)$ with the same length of $m(t)$, otherwise, the computed distance being in both cases the minimum amount of actions needed to reduce $m(s)$ to a string that contains $m(t)$.

We compute the *Sat-delta* value of a formula of the form $s \text{prefix} - \text{of} t$ with respect to a model m by calculating the number of pairwise different characters in $m(s)$ and $m(t)$ up to the length of the shortest of the two strings, and then adding the number of characters (if any) by which $m(s)$ exceeds the length of $m(t)$.

We compute the *Sat-delta* value of a formula of the form $s \text{suffix} - \text{of} t$ with respect to a model m by reversing both $m(s)$ and $m(t)$, and then referring to the computation defined for the operator $\text{prefix} - \text{of}$.

We compute the *Sat-delta* of a formula of the form $s \text{matchr} e$ with respect to a model m by instantiating the concrete string $m(s)$: *Sat-delta* =0 if $m(s)$ matches the regular expression re, 1 otherwise. We also experimented with a version of *Sat-delta* that exploits the *agrep* command [32] to measure the extent by which the string s does not match the regular expression re, and we found the approach to expensive to be of practical use.

Operator isint for Mixed Integer/Reals. Formulas in the mixed theory of integers and reals can predicate on integer, real and a combination of both integers and real variables. The theory introduces the $\text{is} - \text{int}(e)$ that returns $true$ if the value of the real expression e is an integer value, $false$ otherwise. We compute the *Sat-delta* value of a formula of the form $\text{is} - \text{int}(e)$ with respect to a model m as $m(e) - \lfloor m(e) \rfloor$, where $\lfloor m(e) \rfloor$ represents the integer truncation of $m(e)$. Thus, *Sat-delta* is zero if $m(e)$ is a real number that represents an exact integer, and corresponds to the decimal part of $m(e)$ otherwise.

Table 2 summarises the rules that *Sat-delta* applies to remove negations in the atomic formulas that depend on the operators in Table 1.

3.3 *Sat-Delta* in *Utopia*

The hash-function *Sat-delta* provides a straightforward way for *Utopia* to sort formulas and efficiently retrieve the candidate formulas that are the closest (in terms of their *Sat-delta* values) to a target formula, the satisfiability of which has to be determined. *Utopia* simply computes the *Sat-delta* value of the target formula, and uses the computed *Sat-delta* value as the index to identify the k formulas with closest *Sat-delta* values to that of the target. Our current prototype of *Utopia* sets k to 10, that is, it selects and tries to reuse the 10 formulas with *Sat-delta* values closest to the target. In line with the metaphor illustrated in Fig. 2, *Utopia* hypothesises that these 10 formulas are the ones that most likely share solutions with the target formula. *Utopia* then crosschecks the retrieved solutions against the target formula, to determine the satisfiability value of that formula.

4 Reusing Unsat-Cores

This section presents the hash-function *Unsat-footprint* that *Utopia* exploits to efficiently search a repository of previously solved unsatisfiable formulas for an *unsat-core* that proves the unsatisfiability of a new formula. *Unsat-footprint* instantiates the hash-function h_U of the general definition of the *Utopia* algorithm given in Algorithm 1. *Utopia* builds on the common practice that popular SMT solvers return a proof of unsatisfiability of an unsatisfiable formula in the form of an unsat-core, which is a subset of the clauses of the formula whose conjunction is unsatisfiable. *Utopia* stores and reuses unsat-cores to prove that new formulas are unsatisfiable, by retrieving a previously computed unsat-core that proves the unsatisfiability of the new formula.

Unsat-footprint heuristically maps the clauses of each available unsat-core to a fixed size bit vector. It maps each clause to a specific bit, which is then set to 1, while all remaining bits are set to 0. The computation of the hash-function guarantees that the same clauses always map to the same bits, thus allowing *Utopia* to check the containment of an unsat-core in a formula by comparing the *Unsat-footprint* value of the unsat-core with the *Unsat-footprint* value of the formula. For instance, if we have an unsat-core u and two formulas f_1 and f_2 such that:

$$Unsat-footprint(u) = 0101\ldots(\text{with zeros as bit suffix}),$$
$$Unsat-footprint(f_1) = 0001\ldots(\text{with any bit suffix}),$$
$$Unsat-footprint(f_2) = 1101\ldots(\text{with any bit suffix}),$$

then we can deduce that the formula f_1 does not contain the unsat-core u, because u includes a clause that maps to the second bit of the corresponding *Unsat-footprint*, while f_1 does not. Conversely, f_2 might contain u, because f_2 includes two clauses that map to the second and fourth bits of the corresponding *Unsat-footprint*, exactly as the two clauses of u, and thus the two clauses of f_2 might be also in u.

Following the general approach that we illustrated in Section 2, *Utopia* uses the *Unsat-footprint* heuristic to efficiently identify the unsat-cores *close* to a new formula, that is, the unsat-cores composed only of clauses that the formula might contain as well, distinguishing these from the unsat-cores *far* from the formula, because they contain clauses that do not belong to the formula. The experiments reported in Section 5 indicate that the set of unsat-cores close to a formula according to the hash-function *Unsat-footprint* is often very small, thus indicating that *Unsat-footprint* can efficiently identify the reusable unsat-cores, if any exists.

The definition of *Unsat-footprint* exploits the *Bloom filter* data structure [4]. In the remainder of this section we formalise the definition of the hash-function *Unsat-footprint*, and present how *Utopia* exploits *Unsat-footprint* to efficiently search reusable unsat-cores throughout large repositories of previously solved unsatisfiable formulas.

4.1 *Unsat-Footprint*

Function *Unsat-footprint* heuristic is a Bloom filter ([4]) that uses a single hash-function to represent the set of clauses of an unsat-core. Each bit set to 1 in the Bloom filter represents a clause that belongs to the unsat-core.

The *Unsat-footprint* function maps the abstract syntax tree of a clause to an integer number between 0 and $N-1$, where N is the Bloom filter size, by visiting the abstract syntax tree of the clause in depth-first order, and processing the nodes in post-order during the visit. At each node of the abstract syntax tree, the *Unsat-footprint* function (i) computes a local hash code that depends on the entity represented by the node, and (ii) recursively combines the local hash code with the hash codes computed at the children nodes. We compute the local hash-function code depending on the nature of the nodes:

> **Leaf nodes that represent constants:** We hash the value of the constant: For integer constants, we apply the Knuth multiplicative hashing scheme ($hash(n) = n * 2^{32}$) to the value of the constant [21]. For all other constants, we rely on the `hashCode` method of the Java standard library, such as `String.hashCode` for string constants.
>
> **Leaf nodes that represent variables:** We apply the Knuth multiplicative hashing scheme to the value of the positional index associated with the variable according to the order in which the distinct variables appear in the current clause.
>
> **Non-leaf nodes that represent operators:** The local hash code is a predefined prime number that is distinct across the nodes that represent distinct operators and operands of distinct types.

The hash-function combines the local hash codes with the hash codes computed for the children nodes, by relying on the *hash_combine* algorithm implemented in the Boost C++ library [29].

>At the end of the visit, the hash-function returns the hash value of the root node of the abstract syntax tree modulo N (the size of the Bloom filter): This is the value in the range $[0, N-1]$ that indicates the Bloom filter position that *Unsat-footprint* sets to 1 to represent the visited clause.

This hash-function is efficiently computed by exploiting only additions and bit shifting operations, guarantees that identical clauses are always mapped to the same bit of the Bloom filter, and maps different clauses to different bits with high probability.

We use clause-local positional indices to represent the variables, to increase the probability of matching the clauses of an unsat-core with the ones of formulas that may use different variable names, may list the same clauses in a different order, and may include many other variables in clauses other than the ones that match with the unsat-core. Thus, the hash-function enumerates the distinct variables of the current

clause in the order in which they appear in the clause, and renames each variable as v_i where i is the index of the variable in the enumeration. For instance, the formula $y > x$ is renamed to $v_0 > v_1$.

In the current implementation, *Utopia* builds the *Unsat-footprint* Bloom filter that represents the set of clauses in an unsat-core from an initially empty 512-bit Bloom filter (a vector of 512 bits all initially set to zero), by mapping each clause of the unsat-core into a number $n \in [0, 511]$ computed with the hash-function described above, and by setting the nth bit of the Bloom filter to 1. This process produces a Bloom filter in which the bits set to one represent the clauses in the unsat-core. Our experiments indicate that this hash function produces approximately one colliding pair of hashes per group of a hundred clauses.

4.2 *Unsat-Footprint* in *Utopia*

The heuristic *Unsat-footprint* provides a straightforward way for *Utopia* to efficiently select a small set of candidate unsat-cores that are the closest (in terms of their *Unsat-footprint* values) to some new formula, out of large repositories of unsat-cores of unsatisfiable formulas solved in the past. To this end, *Utopia* associates the unsatisfiable formulas in the repository with a dictionary that maps the unsat-cores of the formulas to their corresponding *Unsat-footprint* values.

Utopia checks whether a target formula might contain an available unsat-core, by calculating the *Unsat-footprint* value of the target formula, and by comparing this value with the *Unsat-footprint* values of the unsat-cores in the repository.

Utopia efficiently scans the repository of unsat-cores by obtaining the bitwise complement of the *Unsat-footprint* value of the target formula, and then computing the bitwise-and between this complemented *Unsat-footprint* and the *Unsat-footprint* of the unsat-cores in the repository.[5] When this operation yields zero, indicating the presence of a candidate unsat-core for the new formula in the repository, *Utopia* adds the corresponding unsat-core to a list of candidate unsat-cores that it will later check against the target formula, up to selecting a maximum number (n_U) of candidate unsat-cores. Our current prototype of *Utopia* sets n_U to 10, that is, it always selects and tries to reuse the 10 unsat-cores closest to a new formula in terms of their *Unsat-footprint* values.

The selection requires only bitwise comparisons of integers to filter the candidate unsat-cores. The experimental results reported in Section 5 indicate an overhead below 1 ms for selecting 10 candidate unsat-cores. The experiments reported in Section 5 were conducted on repositories with up to several tens of thousands of unsat-cores. It is possible to efficiently deal with repositories with millions of unsat-cores by suitably indexing the Bloom filters of the unsat-cores in the repository, for example, by using a prefix trie as done in the algorithms of Papalini et al. [27].

Once selected a set of candidate unsat-cores, *Utopia* checks if any of the candidate unsat-cores is contained in the target formula. If so, *Utopia* reports a cache hit and returns a reusable solution that proves that the target formula is unsatisfiable; *Utopia* reports a cache miss otherwise.

4.3 Bloom Filters

A Bloom filter is a fixed-size bit vector that represents a set of elements by relying on a collection of hash functions, each mapping an element of the set to a position in the bit vector [4]. Determining if an element e belongs to a set S amounts to computing the Bloom filter of e, and comparing it to the Bloom filter of S. Hash collisions produce false positives, whose frequency depends on the amount of both bits and hash-functions.

In the specific case of *Unsat-footprint*, large Bloom filters support fine mappings between formula clauses and Bloom filter bits, and large sets of hash-functions further disperse the mapping of distinct sets of clauses on different combinations of the available bits, thus reducing the likelihood that the *Unsat-footprint* of an unsat-core may collide with the *Unsat-footprint* of a formula that does not contain that unsat-core. The advantage of large filters and large sets of hash-functions comes with memory cost for storing the Bloom filters and overhead of computing the hash functions.

Our current implementation of *Unsat-footprint* uses 512-bit Bloom filters and relies on the single hash-function of Section 4.1. With this implementation we observed a false positive rate of 0.41 across the experiments that we report in this paper. *Utopia* is designed to tolerate many false positives: by design it can tolerate up to 9 false positives when selecting 10 candidate solutions for a formula for which there exists a reusable solution indeed. This is why the experiments that we discuss in the next section indicate that *Unsat-footprint* effectively finds a reusable unsat-core for most formulas for which a reusable unsat-core was available indeed, despite an observed false positive rate of 0.41.

The overall false positive rate of 0.41 is due not only to hash collisions in the Bloom filters, but also to the heuristic nature of *Unsat-footprint*. *Unsat-footprint* produces false positives regardless of hash collisions in the Bloom filter, because it indicates potential correspondence between the clauses of formula and unsat-cores independently of the possible mutual relations between the variables referred in the clauses. For example, *Unsat-footprint* would indicate a (one by one) matching between the clauses of the unsat-core $x > y \wedge y \geq z \wedge z > 3x$ and the clauses of the formula $a > b \wedge c \geq d \wedge e > 3f$, because the renaming of the variables of each clause as v_0 and v_1 indicates that the three clauses of the formulas individually match the three clauses of the unsat-core. *Utopia* would discard this solution in the checking phase, because the unsatisfiability of the unsat-core depends on the mutual relations between variables x, y and z in the three clauses, while variables a, b, c, d, e and f are mutually independent in the three clauses of the formula (which in fact is satisfiable in this example).

5. By complementing the *Unsat-footprint* value of a formula, that is, by switching each zero to one and each one to zero, we obtain a Bloom filter that represents all the clauses that are not in the set represented by the original Bloom filter. It follows that a set of clauses A representing a formula likely contains another set of clauses B representing an unsat-core, if and only if the bitwise-and of the complement of the signature of A and the signature of B yields zero. Consider for instance a formula containing four clauses for which our hash function yields the hashes $\{2, 3, 5, 8\}$ and an unsat-core for which our hash function yields the hashes $\{2, 5\}$. The *Unsat-footprint* of the formula and unsat-core would be $[01101001000 \ldots 0]$ and $[01001000 \ldots 0]$, respectively. In this case, the complement of the signature of the first set is the bit vector $[10010110111 \ldots 1]$, which compactly represents all clauses that do not yield the hashes $\{2, 3, 5, 8\}$. The bitwise-and of such bit vector and the *Unsat-footprint* of the unsat-core returns zero indeed, confirming that the clauses with hashes $\{2, 5\}$ might be contained in the set of clauses with hashes $\{2, 3, 5, 8\}$.

TABLE 3
Experiment Summary

Logic	Experiments	Formulas	Atomic predicates per formula: min	q1	med	q3	max	Sat	Unsat
QFLIA	22	798171	1	20	24	39	530	67%	33%
QFNRA	16	28 570	2	12	19	31	50	74%	26%
QFABV	99	181 899	1	3	3	5	1799	70%	30%
QFMIX	2	8120 835	1	2	4	7	121	20%	80%
QFS	1	57 525	1	2	5	21	1452	78%	21%

Disaggregating the false positives that are or are not due to hash collisions requires a non-trivial comparison between the formula and the unsat-core, which is not implemented in the current prototype of *Utopia*.

In our experiments, we experimented with Bloom filters greater than 512 bits without observing significant improvements, possibly because using a single hash-function does not suffice to optimally exploit the additional bits. Experimenting with multiple hash-functions would require simply extending the hash-function presented in Section 4.1 with a parametric seed value, as computing the function for different values of the seed would produce different bit mappings of the formula clauses. Experiments with multiple hash-functions would make sense only with a parallel implementation that avoids the overhead of serially computing the hash-function multiple times for a formula. Investigating the impact of multiple hash functions computed in parallel is part of our future plans.

5 Evaluation

In this section we discuss the results of a set of 140 experiments that assess the effectiveness of *Utopia* to improve the efficiency of different types of symbolic analysis that rely on constraint solving. In each experiment, we consider a program and a symbolic analyser, collect the set of formulas that the analyser submits to the constraint solver during the analysis of the program, and evaluate the effectiveness of *Utopia* to improve the efficiency of the constraint solving step by reusing the solutions across those formulas. All experiments include both satisfiable and unsatisfiable formulas, to evaluate the mutual contributions of the heuristics *Sat-delta* and *Unsat-footprint* that *Utopia* uses to identify the reusable models and the reusable unsat-cores, respectively. The set of experiments as a whole encompasses formulas produced with program analysers that refer to several different logics, thus allowing us to control for the potential sensitivity of *Utopia* with respect to formulas expressed in the different logics.

Overall, the results of the experiments provide empirical evidence that *Utopia* succeeds in identifying reusable solutions for significant portions of the formulas considered in the experiments, and that, by doing so, it results in relevant improvements of the efficiency of the overall constraint solving phase. The experiments show the effectiveness of *Utopia* both in absolute terms and in comparison with the other existing approaches that exploit the structure of the formulas to reuse solutions across formulas that are either equivalent to each other or related by implication.

Below we introduce the subjects that we used in the experiments, discuss the research questions addressed in the experiments, present the prototype implementation of *Utopia* and the experimental setting that we used in the experiments to address the research questions, comment on the results for each research question, and discuss the threats to the validity of our experiments and the countermeasures that we adopted to mitigate the impact of those threats.

5.1 Subjects

We experimented with a total of 140 subjects that encompass sets of formulas from (i) quantifier-free linear integer arithmetic logic, hereafter `QFLIA`, (ii) quantifier-free non linear real arithmetic logic, `QFNRA`, (iii) quantifier-free logic over the theory of bit-vectors and bit-vector arrays, `QFABV`, (iv) quantifier-free logic over the mixed theory of integers and reals, `QFMIX`, and (v) quantifier-free logic over the theory of strings, `QFS`.[6]

Table 3 indicates the considered logics (column *Logic*), the number of experiments that refer to that logic (column *Experiments*), the number of formulas involved in the experiments (column *Formulas*), the distribution of the atomic predicates per formula (columns *Atomic predicates per formula*) that includes minimum (column *min*), first quartile (column *q1*), median (column *med*), third quartile (column *q3*) and maximum (column *max*), and the relative amounts of satisfiable and unsatisfiable formulas (columns *Sat*) and *Unsat*, respectively).

We experimented with (i) 798,171 `QFLIA` formulas that we produced with the symbolic executors *Crest* [10] and *JBSE* [7] during the analysis of 22 C and Java programs, with size from 32 to 10,000 lines of code [13], (ii) 28,570 `QFNRA` formulas that we produced with the invariant generator *Gk-tail* during the analysis of 16 Java classes of the GraphStream and Guava libraries, with size from 22 and 418 lines of code [25], (iii) 8,120,835 `QFMIX` formulas generated with the symbolic executor *JBSE* during the analysis of two Java programs, Gantt and Closure, with 404 and 7,766 lines of code, respectively, that are distributed in the replication package of [6], (iv) 181,899 `QFABV` formulas generated be running the symbolic executor *Klee* for one minute on each of the 99 Coreutils programs of the Linux kernel, with size from 3,307 and 5,217 lines of code [11] (we considered only the formulas that escape the solution caching component included in *Klee* itself, to investigate if *Utopia* can extend the reuse of formulas beyond *Klee*) (v) the 57,525 `QFS` formulas of the Kaluza benchmark, a benchmark of formulas generated with the Javascript symbolic executor *Kudzu* to detect vulnerabilities in web applications [30], for which we do not have information on the size of the programs used to extract these formulas.

The 38 `QFLIA` and `QFNRA` experiments were part of the experimental evaluation that we presented in a previous paper [3], while the other 102 experiments are an original contribution of this paper, and aim to assess the preliminary evidence reported in [3].

5.2 Research Questions

Our experiments address three main research questions.

6. In the paper we report the cumulative results for the groups of experiments that refer to formulas in each logic. We provide a data replication package that includes the detailed results of each single experiment as supplementary material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2019.2898199. Our publicly distributed tool, Julia [1], allows to replicate the experiments.

RQ_1: To what extent does Utopia *identify reusable solutions for the formulas generated during the analysis of a program?*

RQ_1 addresses our hypothesis that the *Sat-delta* and *Unsat-footprint* heuristics identify reusable solutions for satisfiable and unsatisfiable formulas. The larger the amount of formulas for which *Utopia* successfully identifies a reusable solution, the more we may expect to benefit from *Utopia* by reusing those solutions in place of executing the constraint solver to solve those formulas.

We answer RQ_1, by measuring the amount of formulas for which *Utopia* identifies a reusable solution among the formulas submitted to the constraint solver during the analysis of a program, repeating such measurement across our experiments with that consider sets of formulas that were synthesized for many different programs and with program analyzers that refer several different logics. We evaluate the effectiveness of *Utopia* by comparing the reuse data obtained with *Utopia* with respect to the best possible reuse data that could be achieved with the optimal (though inefficient) approach of testing each new formula against all currently available solutions. We consolidate the statistical evidence that there is indeed a causal link between the observed reuse data and the heuristics *Sat-delta* and *Unsat-footprint*, by repeating the experiments with a placebo version of *Utopia*. The placebo version selects the same amount of candidate solutions as *Utopia* at random from the set of available solutions. We refer to the (paired, one tail) Wilcoxon test to confirm our hypothesis that *Utopia* can reuse more solutions than the placebo version across our experiments.

RQ_2: Does Utopia *improve the efficiency of program analysis?*

RQ_2 addresses our hypothesis that *Utopia* decreases the constraint solving costs in program analysis. We answer RQ_2 by quantifying the time saved by reusing solutions.

The time spent for reusing a solution with *Utopia* comes from the time spent for calculating the heuristics *Sat-delta* and *Unsat-footprint* for the formula, the time spent for selecting a set of candidate solutions out of the available ones, and the time spent for testing the formula against the candidate solutions to validate if any of them can be reused indeed. The saving of *Utopia* (with respect to always invoking a constraint solver) depends on the ratio between the formulas for which *Utopia* identifies a reusable solution, thus saving solver calls, and the formulas for which *Utopia* does not identify reusable solutions, for which it pays both the overhead of the reuse process and the cost of having to call the solver anyway. In the best case, *Utopia* always finds a reusable solution, thus reducing the time spent to invoke a solver to the time needed to identify the solution and check that is it a valid solution indeed. In the worst-case, *Utopia* does not identify any reusable solution, thus wasting the overhead of searching candidate solutions and proving that they are not valid solutions for the target formulas indeed.

RQ_2 aims to disprove the null-hypothesis that *Utopia* results in penalising the efficiency of the program analysis, because it is either slower than computing the solutions directly with the solver, or results in too high overhead on the overall constraint solving time.

We answer RQ_2 by measuring and comparing the time to complete all the constraint solving tasks issued by a program analyzer with and without *Utopia*. When working with *Utopia*, we measure the time of the constraint solving task as the sum of the time spent to execute *Utopia*, and the time spent to solve the formulas for which *Utopia* cannot identify any reusable solution.

RQ_3: Does Utopia *outperform its competing approaches?*

RQ_3 addresses the hypothesis that *Utopia* identifies more reusable solutions than state-of-the-art approaches. The distinctive characteristic of *Utopia* is that it does not depend on the logical structure of the formulas, but evaluates the formulas against the reference solutions, and explicitly tests them against a heuristically selected set of available solutions, while state-of-the-art approaches exploit the logical structure of the formulas to identify formulas that are either equivalent or related by implication. This distinctive characteristics widen the range of reuse opportunities of *Utopia* with respect to state-of-the-art approaches, because *Utopia* can reuse solutions also among formulas that are neither equivalent nor related by implication. On the other side, as discussed in Section 2, *Utopia* may suffer from false negatives, thus the effectiveness of *Utopia* with the respect to approaches based on structural equivalence of formulas shall be studied empirically rather than demonstrated analytically.

Additionally, while *Utopia* is to a large extend logic independent, and we have instantiated approach for the logics listed in Table 1, state-of-the-art approaches strongly depend on the logic, and target only quantifier-free linear integer arithmetic logic. Thus we answer $RQ3$ by comparing the reuse rate of *Utopia* with respect to the reuse rate of state-of-the-art approaches in the context of quantifier-free linear integer arithmetic logic, which involves 22 out of our 140 experiments.

5.3 Prototype

We executed the experiments with *Julia*, a prototype version of *Utopia* [1]. *Julia* is implemented in Java, and works as the front-end layer of a back-end constraint solver set at configuration time. From the caller viewpoint, *Julia* acts as a constraint solver, that is, it takes as input a formula, and returns either a model proving that it is satisfiable, or an unsat-core proving that it is unsatisfiable. From the internals viewpoint, *Julia* caches the solutions that the back-end solver computes, and exploits the *Utopia* technique to reuse them when possible.

When called with a formula, *Julia* separates the input formula into mutually independent conjunctive sub-formulas,[7] and then it (i) applies the *Utopia* technique to determine either the satisfiability or the unsatisfiability of the individual sub-formulas by reusing either models or unsat-cores identified with the *Utopia* approach, (ii) calls the constraint solver for the sub-formulas for which it does not find a reusable solution, and (iii) aggregates the results of the sub-formulas to produce a solution of the original formula.

Julia first determines the satisfiability of each sub-formula by (i) calculating the *Sat-delta* value of the formula as the average *Sat-delta* with respect to the reference models presented in Table 4, (ii) selecting the ten models that

7. The process of decomposing a conjunctive formula into the set of its mutually independent sub-formulas is known as *formula slicing* [11], and is generally adopted by state-of-the-art formula caching approaches. Two sub-formulas are mutually independent if they include conjuncts of the original formula that do not have common variables.

TABLE 4
Reference Models in the Prototype *Julia*

Logic	Reference models
• QFLIA, QFNRA	• $[\forall v \mid m(v) = 0]$ • $[\forall v \mid m(v) = 100]$ • $[\forall v \mid m(v) = -1000]$
• QFABV	• $[\forall v \mid m(v) = 0 \ldots 0$ • $[\forall v \mid m(v) = 0 \ldots 01]$
• QFMIX	• $[\forall v \mid m(v) = 0]$ • $[\forall v \mid m(v) = -2^{32}]$ • $[\forall v \mid m(v) = 2^{32} - 1]$
• QFS	• $[\forall v \mid m(v : str) = "", m(v : int) = 0]$ • $[\forall v \mid m(v : str) = "a", m(v : int) = 1]$

Julia *computes the* Sat-delta *value of a formula in a logic (column* Logic*) as the average of the* Sat-delta *values with respect to a set of reference models (column* Reference models*) for that logic.*

correspond to the (previously solved) satisfiable formulas whose *Sat-delta* values are the closest to the one of the target formula, and (iii) testing the target formula against such models to determine whether any of them can be reused to prove the target formula satisfiable.

If *Julia* finds a reusable model, it logs a cache hit and returns the identified solution, otherwise, *Julia* repeats the search procedure with the *Unsat-footprint* heuristic to identify the ten available unsat-cores that are most likely contained in the target formula, and tests whether any of these unsat-cores is contained in the target formula. If *Julia* finds an unsat-core for the formula, it logs a cache hit, and returns the identified unsat-core as a solution that proves that the target formula is unsatisfiable. Otherwise, *Julia* reports a cache miss, solves the formula with the chosen SMT solver, and caches the solution in the relevant repository.

Sat-delta works with any set of reference models. To chose the reference model in the *Julia* protoype, we experimented with different sets of models, composed of one, three, five and ten models for each of the considered logic, and we did not notice significative differences in the results. We adopted the models that result in the best trade-off between reuse opportunities and efficiency.

Julia can be used with a variety of back-end SMT solvers. In our experiments, we used Microsoft *Z3* version 4.5.1, the latest version of *Z3* available at the time of our experiments [15]. In the QFS experiment, we used *Z3-Str*, an extension of *Z3* which specifically addresses formulas from the theory of strings [33] because *Z3-Str* is more efficient of these formulas.

5.4 Experimental Setting

In our experiment, we compute the solutions of each given set of formulas both with *Julia*, instantiated with *Z3* (*Z3-Str* for formulas on strings), and directly with *Z3* (*Z3-Str*), to measure the improvement of *Utopia* over the plain use of a constraint solver. We conducted a total of 140 experiments with sets of formulas from five logics, as summarised in Table 3.

We executed the QFMIX experiments with the *JBSE* symbolic executor that uses *Z3* as default solver, and that we extended with *Julia*. We symbolically analysed the subject programs Gantt and Closure with *JBSE*, with and without *Julia*, thus we directly quantify the impact of *Utopia* on symbolic execution. We executed the QFLIA, QFNRA, QFABV and QFS experiments with *Crest*, *Gk-tail*, *Klee* and *Kudzu*, respectively. Since none of these tools is integrated with *Julia*, we collected the formulas produced while analysing the subject programs, and processed these formulas offline with both *Julia* and *Z3*. Thus we quantify the impact of *Utopia* with respect to the constraint solving activities in the program analysis sessions.

In the experiments, *Julia* incrementally populates an initially empty repository with the solutions computed with the constraint solver after each cache miss. Since we use a deterministic version of *Z3*, which always returns the same solution for the same formula, the results of the experiments are deterministic.

In the experiments, we quantify the efficiency of *Julia* and *Z3* in terms of their *execution time*, that is, the time to complete the whole constraint solving task, and compare the running time of *Julia* and the running time of calling *Z3* directly.

We quantify the effectiveness and relevance of the *Sat-delta* and *Unsat-footprint* heuristics by measuring the *reuse-ratio*, that is, the number of formulas that *Julia* solves by identifying a reusable solution by means of the heuristics (the cache hits) over the total number of formulas considered in each experiment (which include both cache hits and cache misses). We compare the reuse-ratio of *Julia*, with the reuse-ratio of *J-exhaustive* and *J-random*, two versions of *Julia* that substitute the heuristics *Sat-delta* and *Unsat-footprint* with simple selection strategies that we refer to as baseline.

J-exhaustive tests the target formulas against all solutions currently available in the repositories of satisfiable and unsatisfiable formulas. Thus, *J-exhaustive* identifies all solution of all formulas that can be solved by reusing the solution of some formulas solved beforehand. *J-exhaustive* is very inefficient, but provides an upper bound to evaluate the effectiveness of the heuristics *Sat-delta* and *Unsat-footprint*. The closer the reuse-ratios of *Julia* are to the ones of *J-exhaustive* across the experiments, the stronger the evidence that the heuristics *Sat-delta* and *Unsat-footprint* approximate the optimal effectiveness.

J-random tests the target formulas against ten models and ten unsat-cores selected at random out of the models and the unsat-cores currently available in the repositories. Thus, *J-random* represents the *palcebo* alternative of *Julia* with respect to using the heuristics *Sat-delta* and *Unsat-footprint*. We can claim that the heuristics *Sat-delta* and *Unsat-footprint* are effective only if we gain statistical evidence that the reuse-ratios of *Julia* significantly outperform the ones of *J-random*.

We run all the experiments on a MacBook Pro equipped with a 2.5 GHz Intel Core i7 processor and 16 GB of RAM. To control for bias due to the execution environment, we execute each run of each experiment on a freshly rebooted machine, repeat each execution ten times, and average the results. This also allows us to account for the intrinsic nondeterminism of *J-random*. We set a timeout of 10 seconds for *Z3* to solve each formula. The timeout never expired in any of our experiments.

The results depend on the order in which *Julia* processes the formulas considered in the experiments. In all but *Kudzu*

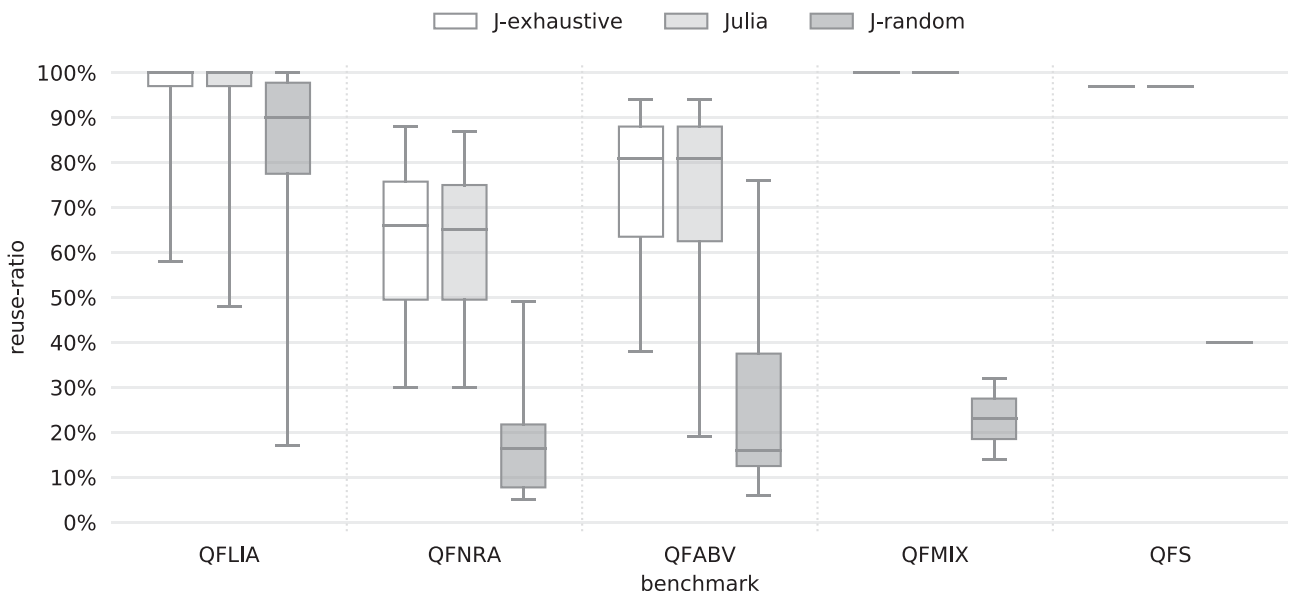

Fig. 4. Reuse-ratio of *Julia*, *J-exhaustive* and *J-random* for both satisfiable and unsatisfiable formulas.

experiments, we process formulas in the order in which they were originally produced, to match a realistic scenario for the application of our approach. Since *Kudzu* does not process formulas in a specific order, we executed the *Kudzu* experiments by both reversing and randomly rearranging the order of formulas, to study the impact of the order of evaluation, without observing any significant difference in the results.

5.5 RQ_1 – Effectiveness

RQ_1 addresses the effectiveness of *Utopia* in identifying reusable solutions by means of the underlying heuristics *Sat-delta* and *Unsat-footprint*. We answer this research question by comparing the reuse-ratios that we measured with the prototypes *Julia*, *J-exhaustive* and *J-random* in 140 experiments.

Fig. 4 shows the box-plot charts that summarise the distribution of the reuse-ratios across all experiments for the different logics. Figs. 5 and 6 show the box-plots for satisfiable and unsatisfiable formulas, respectively, to illustrate the individual contributions of the heuristics *Sat-delta* and *Unsat-footprint*. The figures show three box-plots that correspond to the reuse-ratios of *Julia*, *J-exhaustive* and *J-random*, respectively, for each of the five logics considered in our experiments, `QFLIA`, `QFNRA`, `QFABV`, `QFMIX` and `QFS`. The box-plots highlight the minimum, first quartile, median, third quartile and maximum value of the reuse-ratios observed in the corresponding group of experiments.[8]

The box-plots in Figs. 4, 5 and 6 show that the reuse-ratios of *Julia* consistently outperform the reuse-ratios of the random (placebo) strategy, and are extremely close to the reuse-ratios of the (ideal) exhaustive strategy.

8. We remark that the minimum of 48 percent that *Julia* achieves for the experiments with all `QFLIA` formulas (Fig. 4) is indeed consistent with the minimum of 0 percent that it achieves with reference to only either satisfiable or unsatisfiable `QFLIA` formulas (Figs. 5 and 6). In fact, the minimum reuse of satisfiable (resp. unsatisfiable) formulas is achieved for a program for which the considered formulas include a small set of satisfiable (resp. unsatisfiable) formulas that do not share solutions among them, whereas for the same program there are many unsatisfiable (resp. satisfiable) formulas among which *Julia* successfully identifies a good portion of reusable solutions.

We gather statistical evidence that *Julia* is considerably better than *J-random* and very close to *J-exhaustive* with the (paired, one tail) Wilcoxon test. The Wilcoxon test supports the claim that the reuse-ratio of *Julia* outperforms the reuse-ratio of *J-random* by 38, 42 and 24 percent when considering all formulas, only satisfiable formulas and only unsatisfiable formulas, respectively, with p-values of 0.016, 0.007 and 0.02 in the three cases, respectively. It supports the claim that the reuse-ratio of *Julia* differs by less than 0.6, 0.5 and 1.5 percent from the best values computed with *J-exhaustive* in the three cases, respectively, with p-values of 0.009, 0.022 and 0.026, respectively.

The limited difference between *Julia* and *J-random* box-plots in some `QFLIA` experiments depends on the characteristics of the case study. In fact, many of the 22 `QFLIA` experiments contain large sub-sets of equivalent formulas that share solutions, as evident from the upper bound of the reuse-ratio that is very close to 100 percent as median value in the boxplot of *J-exhaustive* in Fig. 4. As a consequence, these experiments tend to experience few cache misses, and thus the solution repositories of these experiments contain small amounts (few tens) of formulas, since in each experiment we populate repositories from scratch with the solutions computed with the solver upon the cache misses, thus making random sampling possibly effective.

Nevertheless, the variance of the box-plots witnesses the steadily effectiveness of *Julia* across all the `QFLIA` experiments, in contrast with the large variability of *J-random* that performs badly for the subset of the experiments with many formulas and few equivalent formulas. The results of the `QFMIX` and `QFS` experiments repotted in Fig. 4 further strengthen this observation: In these experiments the limitations of *J-random* are evident, while *Julia* is indeed effective even though the upper bound of the reuse-ratio measured with *J-exhaustive* is also very close to 100 percent.

The data in Table 5 indicate that the reuse rate of *Utopia* does not depend on the size of the formulas. The table reports the reuse-ratio of *Julia* across the formulas in the different logics, for (i) the entire sets of satisfiable and unsatisfiable

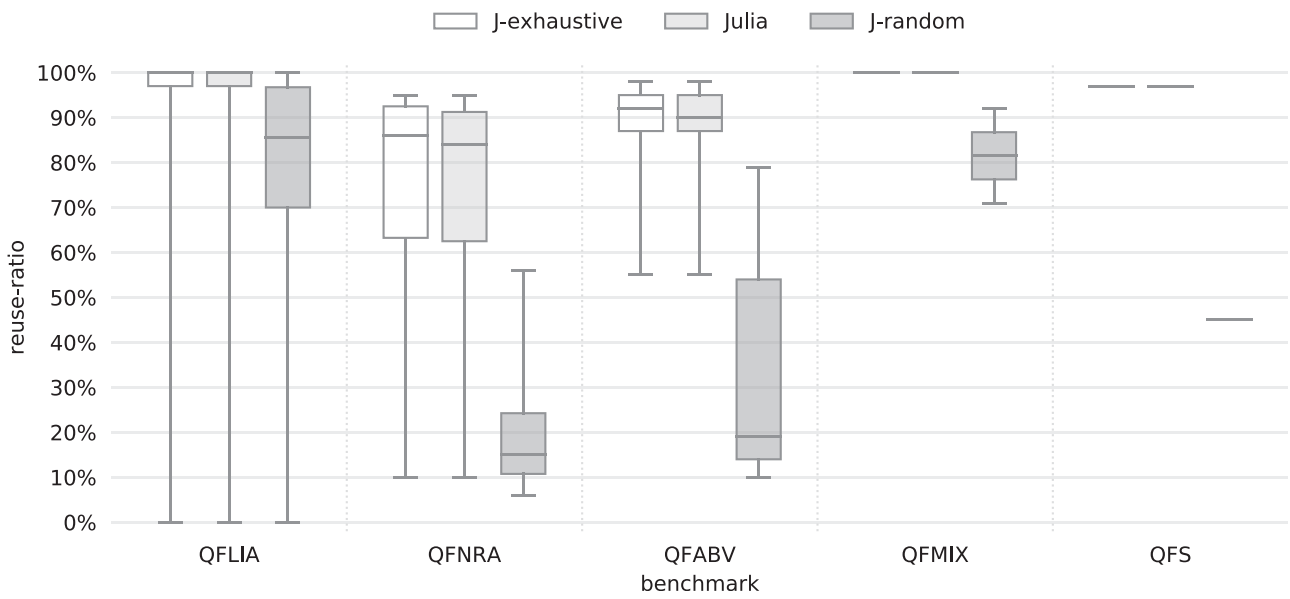

Fig. 5. Reuse-ratio of *Julia*, *J-exhaustive* and *J-random* with only satisfiable formulas.

formulas (columns *Reuse-ratio for satisfiable/unsatisfiable formulas, considering: all formulas*), (ii) the set of formulas with no more and (iii) the set of formulas with more atomic predicates than the median number of atomic predicates in the corresponding set (columns *Reuse-ratio for satisfiable/unsatisfiable formulas, considering: only small/large formulas*).

The even reuse ratio across the sets of formulas for all logics indicates that the reuse of *Utopia* does not depend on the size of the formulas: We do not observe significant differences in the reuse rate of *Utopia* for small and large formulas across all logics, but satisfiable QFABV formulas, for which *Julia* found a reusable solution for the 84 percent of the small formulas, but only for the 59 percent of the large formulas.

Overall, our experiments support a positive answer to research question RQ_1, confirming the ability of the *Utopia* approach to successfully steer the identification of reusable solutions both for satisfiable and unsatisfiable formulas.

5.6 RQ_2 – Efficiency

RQ_2 addresses the impact of *Utopia* on the efficiency of program analysers. We answer this research question by comparing the execution time of *Julia* with respect to both the execution time of *Z3* (version 4.5.1) and *J-random*, across 140 experiments. The first set of results (*Julia* with respect to *Z3*) evaluates the impact of *Utopia* on the efficiency of state-of-the-art program analysers, which rely on suitable constraint solvers. The second set of results (*Julia* with respect to *J-random*) evaluates the contribution of the *Utopia* heuristics to achieve the efficiency gain.

Fig. 7 compares the execution time of *Julia*, *J-random* and *Z3*, plotting the cumulative execution time of all the experiments for each logic. The bars in the figure report the running time as a portion of the running time of *Z3*, which is the 100 percent reference, and are annotated with the running time in seconds.

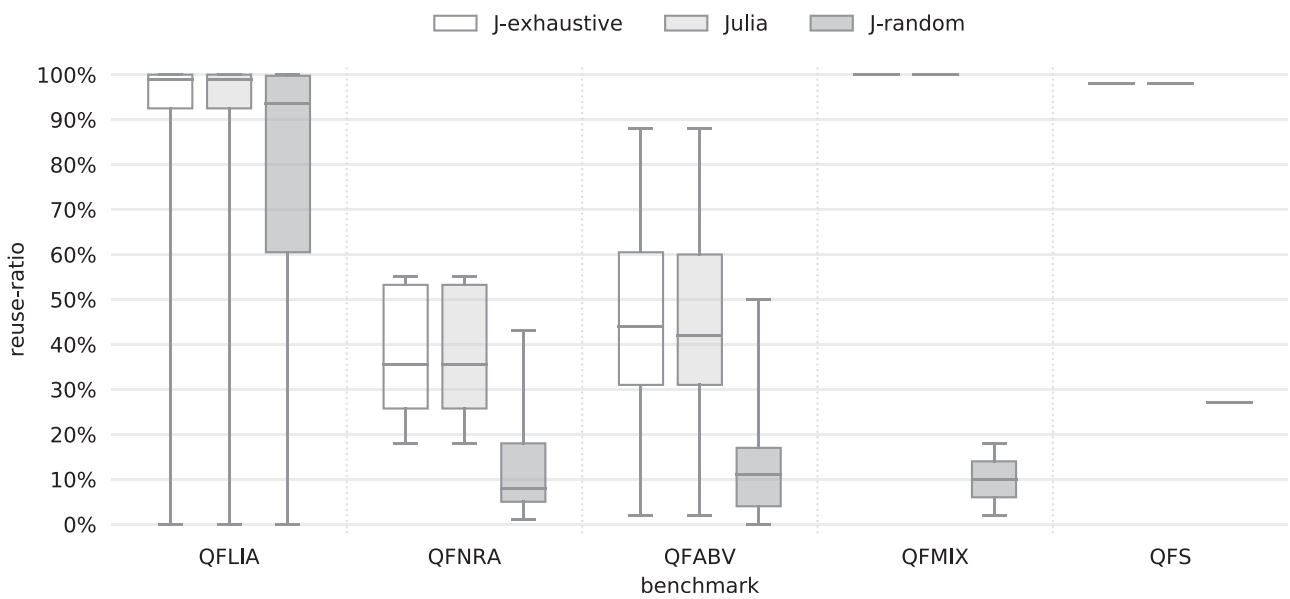

Fig. 6. Reuse-ratio of *Julia*, *J-exhaustive* and *J-random* with only unsatisfiable formulas.

TABLE 5
Sensitivity of *Utopia* with Respect to the Size[§] of the Formulas

	Reuse-ratio for satisfiable formulas, considering:			Reuse-ratio for unsatisfiable formulas, considering:		
Logic	all formulas	only small formulas	only large formulas	all formulas	only small formulas	only large formulas
QFLIA	99,99	99,99	99,93	99,87	99,96	99,70
QFNRA	91,48	89,18	93,79	37,15	33,73	41,08
QFMIX	99,98	99,99	99,96	99,74	99,94	99,53
QFABV	72,83	84,19	59,12	32,59	32,51	32,68
QFS	96,85	98,51	94,73	98,25	98,35	98,15

[§]*Small and large formulas are formulas with no more or more atomic predicates than the median number of atomic predicates across all the formulas in the corresponding set, respectively.*

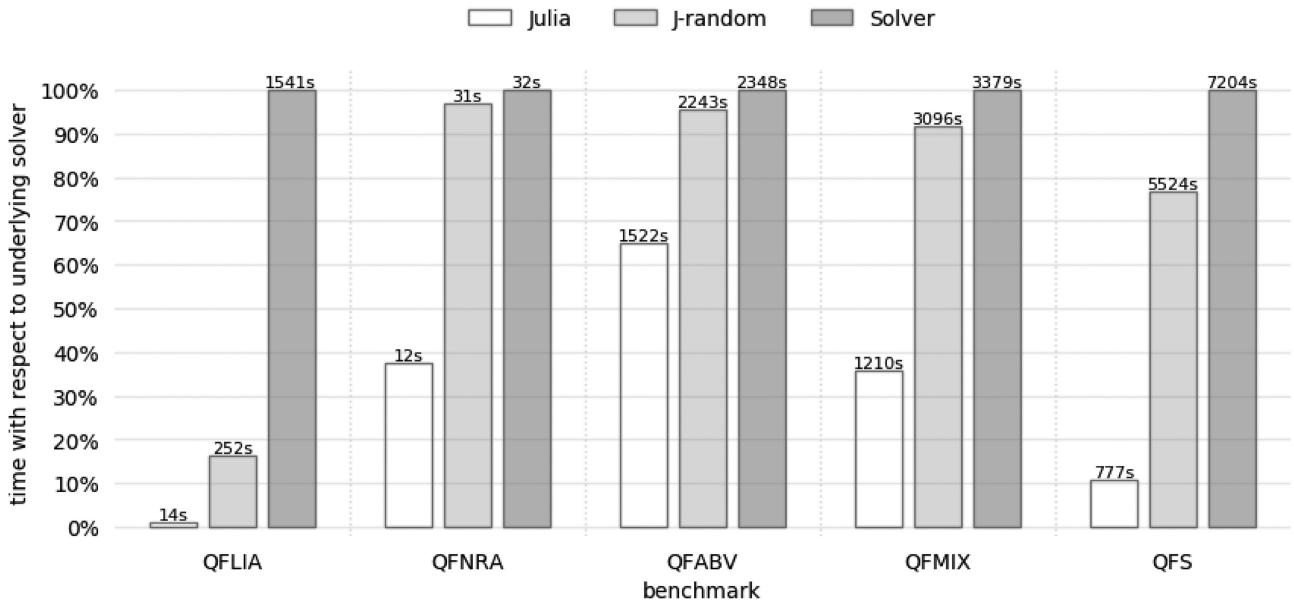

Fig. 7. Execution time of *Julia* and *J-random* with respect to *Z3*.

Fig. 7 indicates that *Julia* reduces the constraint solving time from 32 to 99 percent with respect to *Z3*. The saving time ranges from 20 seconds for the fast experiments with `QFNRA`, to 1 hour and 47 minutes for the experiments with formulas in the `QFS` logic, where *Julia* solves all formulas in approximately 13 minutes, while *Z3* takes 2 hours. In the particular case of the `QFMIX` experiments, where we executed *JBSE* with either *Julia* or *Z3* as backend constraint solver, *Julia* reduces the analysis time from 56 to 20 minutes with respect to simply using *Z3*, thus *Julia* improves the execution time of *JBSE* by 64 percent.

The comparison with *J-random* further highlights the contribution of *Utopia*. *J-random* produces negligible improvements on *Z3* in all other experiments but the `QFLIA` and `QFS` experiments, and is consistently slower than *Julia* in all experiments, including the `QFLIA` and `QFS` ones, thus confirming the usefulness of effective heuristics.

Table 6 presents the execution time for solving satisfiable and unsatisfiable formulas with *Z3* and *Julia* (columns *Time (sec)* with *Z3* and *Time (sec)* with *Julia*, sub-columns *Sat. formulas* and *Unsat. formulas*, respectively), and the overhead that *Julia* pays over a direct call to the solver (column *Utopia overhead*). The data reported in the table indicate that *Utopia* considerably reduces the time required to solve both satisfiable and unsatisfiable formulas, at the cost of a generally acceptable overhead that ranges between 7 and 117 seconds for the benchmarks considered in our experiments.

Overall, our experiments support a positive answer to research question RQ_2, by confirming that *Utopia* can successfully reduce the costs of constraint solving in program analysis, and showing the impact of the *Utopia* heuristics on the improvements.

5.7 RQ_3 – Competing Approaches

RQ_3 addresses the improvement of *Utopia* over state-of-the-art approaches. We answer this research question by comparing both the reuse-ratio and the running time of *Julia*, with the reuse-ratio and the running time of the state-of-the-art approaches *Green* ([31]), *GreenTrie* ([20]), *Recal* and *Recal+* ([2]).[9] We execute the experiments with the `QFLIA` formulas only, since the state-of-the-art approaches are strongly related to the logics and deal mainly with `QFLIA` formulas.

Fig. 8 shows the box-plots of the reuse-ratios of the different approaches on all `QFLIA` experiments. Each box-plot shows the minimum, first quartile, median, third quartile

9. We used the version of *Green* available at https://github.com/green-solver/green-solver, commit *3e2ce94*, the latest available at the time of writing. *GreenTrie*, *Recal* and *Recal+* have no official releases: we obtained *GreenTrie* by contacting the authors [20], and used the version of *Recal* maintained at our own lab [2].

TABLE 6
Execution Time and Overhead for Satisfiable and Unsatisfiable Formulas

Logic	Time (sec) with Z3		Time (sec) with Julia		
	Sat. formulas	Unsat. formulas	Sat. formulas	Unsat. formulas	Utopia overhead
QFLIA	1,277	264	1	<1	13
QFNRA	27	5	2	4	7
QFABV	760	1,588	299	1,107	117
QFMIX*	441	1,529	1	7	17
QFS §	5,612	953	208	16	3

*We computed the QFMIX data when executing Julia integrated in JBSE. The sum of the time of the QFMIX row is less than the total time indicated in Fig. 7 for the corresponding experiments, because the total time reported in Fig. 7 includes the execution time of the whole JBSE symbolic execution procedure, while the time reported in this table is only the time required for solving the constraints.

§The QFS dataset includes a small amount of formulas for which Z3 is not able to compute the solution, that is, constraints for which Z3 returns unknown. The sum of the time of the QFS row is less than the totals indicated in Fig. 7 for the corresponding experiments, in fact, the time spent on the unknown-solution formulas is slightly greater than 600 seconds in both experiments.

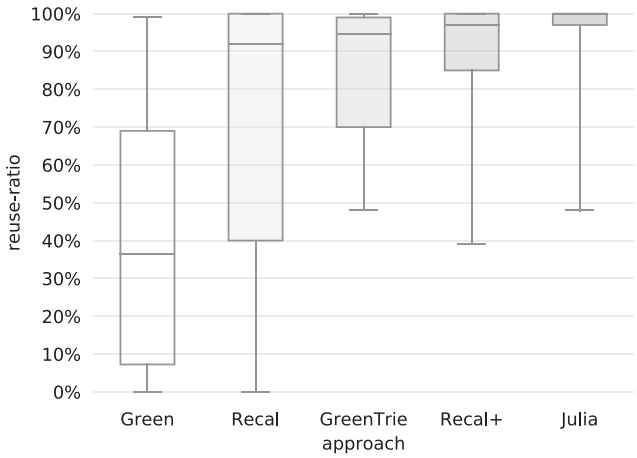

Fig. 8. Reuse-ratio of *Green*, *Recal*, *GreenTrie*, *Recal+* and *Julia* on the QFLIA experiments.

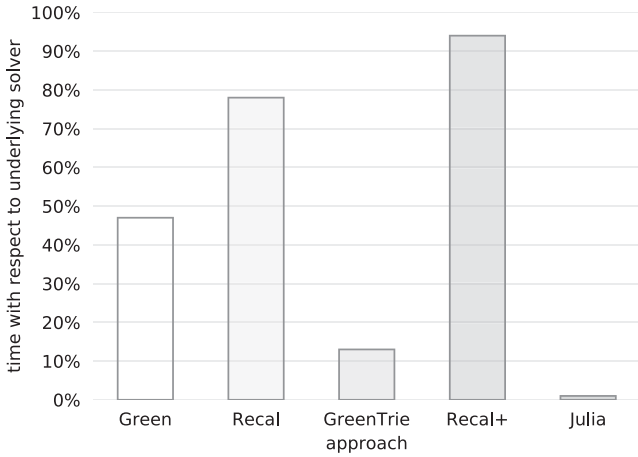

Fig. 9. Running time of *Green*, *Recal*, *GreenTrie*, *Recal+* and *Julia* with respect to *Z3* on the QFLIA experiments.

and maximum reuse-ratios for each approach across the experiments. The *Green* approach presents the lowest reuse-ratios (with a median 37 percent), and *Utopia* the highest ones (with a median greater than 99 percent).

Fig. 9 shows the execution time of all approaches, normalised with respect to the running time of the solver *Z3*, the underlying solver for all considered approaches in these experiments. Different approaches rely on different strategies to invoke *Z3*, either creating a new *Z3* process instance for every formula, or communicating with a single process instance of *Z3* all along an experiment, with the former strategy being in general less efficient than the latter. We carefully addressed this issue by repeating the measurement of the running time of standalone-*Z3* with both invocation strategies, and normalising the execution time of each approach with respect to the time of *Z3* invoked with the strategy that the approach uses.[10]

The data in Fig. 9 confirm that *Julia* is the most efficient approach, with a saving of over 99 percent of the solving time, while the second best approach *GreenTrie* saves approximately 87 percent of the solving time. *Green*, *Recal* and *Recal+* save only 53, 22 and 6 percent of the solving time, respectively.

Overall our experiments indicate that *Utopia* outperforms the competing approaches for the QFLIA logic, and is the only approach that extends to multiple logics.

5.8 Threats to Validity

In this section we discuss the main threats to the validity of the results reported in the paper:

Prototype implementation. We extensively tested our prototype, and verified the correctness of the produced solutions by checking their validity with the Microsoft Z3 SMT solver. We executed the prototype on all the formulas used in the experiments, and verified that the solutions identified as reusable are indeed solutions for the candiate formulas with the Microsoft Z3 SMT solver.

Structure of the formulas. We experimented with formulas produced by analysing a large set of programs with *JBSE*, *Crest*, *Klee* and Kudzu symbolic executors and the *Gk-tail* program analyser. The formulas produced with the symbolic executors share similar structures: They are large conjunctions of homogeneous expressions, that is, expressions in the same logic being it integer, bit-vector or string. The formulas produced with *Gk-tail* are all implications betweens non-linear real expressions. Although we have no elements that suggest the dependence of *Utopia* on specific

10. In the experiments we used *Z3* in the standard configuration, without considering the *incremental-mode*, in which *Z3* attempts to save partial artefacts of the inference process to reuse those artefacts while solving future formulas.

structures of formulas, generalising the results to formulas with different structures requires new experiments.

Impact on program analysis. We conducted the QFMIX experiments with the symbolic executor *JBSE* integrated with *Julia*, thus the results indicate the full efficiency gain of *Utopia* on *JBSE*. We conducted all other experiments by storing the formulas computed while analysing the benchmarks with the corresponding analyzers, and processing these formulas offline with our prototype *Julia*, *Z3* and (in the case of the QFLIA experiments) the other tools. Thus the results indicate the efficiency gain in the formula evaluation, but only approximate the full gain in a realistic usage scenario. As a further observation, we collected the QFABV formulas by executing the symbolic executor *Klee* for a minute on the 99 Coreutils programs (Section 5.1), thus we cannot claim that our results with these formula datasets witness the quantitative efficiency gain that our *Utopia* approach may have in long analysis sessions with *Klee*.

Order of formula evaluation. In our experiments, we fed *Utopia* with formulas in the order they were originally produced by the corresponding program analysis technique to match a realistic scenario for the application of our approach. We investigated the impact of feeding *Utopia* with the same formulas in different order by re-executing the experiments feeding *Utopia* with formulas in both reversed order and rearranged randomly, and we did not reveal noticeable differences in the results. This indicates that the order in which formulas are given to *Utopia* does not impact the results.

6 Related Work

Despite the remarkable progresses of SMT solvers, they still remain one of the main bottlenecks to the scalability of program analysis techniques that rely on them. In the last decade, many different techniques have been developed to cope with this bottleneck, including (i) using many solvers concurrently to mitigate the weaknesses of individual solvers, (ii) complementing SMT solvers with external optimisations, and (iii) reducing the number of queries of symbolic program analysis techniques to SMT solvers.

Palikareva et al. proposed to concurrently execute multiple heterogeneous solvers to improve the running time of symbolic execution. They observed that modern SMT solvers have different strengths and weaknesses and that it is impossible to identify in advance the solver that perform better for each formula [26]. Palikareva et al.'s framework executes multiple solvers in parallel on the same formula, and returns the earliest produced solution. They successfully augmented the *Klee* symbolic executor with their framework, largely improving its scalability.

A different research thread focused on complementing SMT solvers with external optimisations. Erete and Orso proposed a technique that exploits the concrete values produced during dynamic symbolic execution to iteratively simplify the formulas to solve [18], and show that the approach can speed up the verification, but may also be inefficient when iterating too many times.

The approaches most closely related to *Utopia* are techniques that reduce the number of queries of symbolic executors to SMT solvers by reusing solutions across formulas.

The *Klee* symbolic executor includes two caching frameworks, called the *branch cache* and *counterexample cache*, that target formulas in the quantifier-free theory of bit-vectors and bit-vector arrays [11]. The branch cache simply "remembers" formulas and their solutions. The counterexample cache determines whether a target formula contains or is contained in other formulas solved in the past. If the target formula is contained in a satisfiable, respectively unsatisfiable, formula, then it is also satisfiable, respectively unsatisfiable.

Green is a caching framework that targets formulas belonging to the quantifier-free linear integer arithmetic logic [31]. *Green* simplifies a target formula by exploiting a predefined set of simplification rules, and checks if the simplified formula belongs to a database of formulas solved in the past. If the target formula matches a satisfiable, respectively unsatisfiable, formula in the database, then it is also satisfiable, respectively unsatisfiable.

GreenTrie extends the *Green* caching framework by targeting formulas belonging to the quantifier-free linear integer arithmetic logic [20]. *GreenTrie* can identify some particular cases of logical implication between two formulas. *GreenTrie* checks if a target formula implies or is implied by other formulas already 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, then it is also unsatisfiable.

Recal is a caching framework that targets formulas belonging to the quantifier-free linear integer arithmetic logic [2]. *Recal* simplifies a target formula by exploiting a predefined set of simplification rules, encodes the simplified formula as a matrix, and produces a unique canonical form of the matrix representation to efficiently check if the target formula belongs to a database of solved formulas. *Recal*+ extends *Recal* by identifying some particular cases of logical implication between two formulas, similarly to *GreenTrie* [20].

With its distinctive elements *Sat-delta* and *Unsat-footprint*, *Utopia* differs substantially from state-of-the-art approaches, since *Utopia* looks for formulas that share solutions with the given target formula, while the above approaches look for formulas equivalent to or contained in the target formula. In this way, *Utopia* extends reusability beyond equivalent or contained formulas, to a much wider set of formulas that share some solutions, independently from their structural relationships.

Li et al. designed a technique that exploits machine learning to identify a solution (a model) for a formula. Li et al.'s fitness function quantifies the distance of candidate from satisfying models. Li et al.'s fitness function is defined for formulas in the QF-LIA logic, and is indeed compatible with our distance function *Sat-delta* for the QF-LIA logic. The work described in this paper defines *Sat-delta* for many other logics, and could be thus potentially used to extend the technique proposed by these authors to all the logics supported by our approach.

7 Conclusion

In this paper we present *Utopia*, a novel approach that stores and reuses solutions of formulas from many popular logics to determine the satisfiability or the unsatisfiability of new formulas. Contrarily to existing approaches that rely mostly

on syntactical rules and pattern matching to identify logically related formulas to reuse solutions, our approach exploits two heuristics to identify a small set of likely reusable solutions for a formula out of a potentially large set of solutions belonging to formulas solved in the past.

Our experimental evaluation shows that our approach outperforms vanilla solvers and its competing approaches both in terms of the quantity of solutions it can reuse and in terms of the time required to solve large sets of formulas. Moreover, our approach is the only one which can currently deal with formulas from most SMT-LIB theories, while most other approaches are tuned to work with formulas from only one (or very few) logics.

In our previous work we defined the *Sat-delta* heuristic exclusively for the quantifier-free integer and real arithmetic logics. In this paper, we extended the *Sat-delta* heuristic to cope with most standard SMT-LIB theories and proved its effectiveness on a much larger set of benchmarks. We also introduced the novel *Unsat-footprint* heuristic to enable the reuse of unsatisfiable cores and proved both its effectiveness and efficiency.

Our future plans include developing a distributed version of the *Utopia* approach described in this paper to serve the research community. The solutions of the formulas solved by some research groups could then be used to solve the formulas produced by other groups in a fraction of the time required to solve them in the very first place. We believe that, in the long run, this could largely improve the scalability of modern symbolic program analysis techniques.

Acknowledgments

This work is partially supported by the Swiss SNF project CloSE (200021_149997/1) and by the Italian MIUR PRIN project GAUSS (Contract 2015KWREMX).

References

[1] A. Aquino, Julia. [Online]. Available: https://bitbucket.org/andryak/julia. Accessed on: Sep. 11, 2017.
[2] A. Aquino, F. A. Bianchi, M. Chen, G. Denaro, and M. Pezzè, "Reusing constraint proofs in program analysis," in *Proc. Int. Symp. Softw. Testing Anal.*, 2015, pp. 305–315.
[3] A. Aquino, G. Denaro, and M. Pezzè, "Heuristically matching solution spaces of arithmetic formulas to efficiently reuse solutions," in *Proc. Int. Conf. Softw. Eng.*, 2017, pp. 427–437.
[4] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[5] E. Bounimova, P. Godefroid, and D. Molnar, "Billions and billions of constraints: Whitebox fuzz testing in production," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 122–131.
[6] P. Braione, G. Denaro, A. Mattavelli, and M. Pezzè, "Combining symbolic execution and search-based testing for programs with complex heap inputs," in *Proc. Int. Symp. Softw. Testing Anal.*, 2017, pp. 90–101.
[7] P. Braione, G. Denaro, and M. Pezzè, "Symbolic execution of programs with heap inputs," in *Proc. Eur. Softw. Eng. Conf. Held Jointly ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2015, pp. 602–613.
[8] R. Brummayer and A. Biere, "Boolector: An efficient SMT solver for bit-vectors and arrays," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2009, pp. 174–177.
[9] R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich, "The openSMT solver," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2010, pp. 150–153.
[10] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Proc. Int. Conf. Automated Softw. Eng.*, 2008, pp. 443–446.
[11] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. Symp. Operating Syst. Des. Implementation*, 2008, pp. 209–224.
[12] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013.
[13] M. Chen, "Reusing constraint proofs in symbolic analysis," Phd Thesis, Università della Svizzera Italiana (USI), 2018.
[14] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The MathSAT5 SMT solver," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2013, pp. 93–107.
[15] L. De Moura and N. Bjorner, "Z3: An efficient SMT solver," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2008, pp. 337–340.
[16] L. De Moura and N. Bjorner, "Satisfiability modulo theories: Introduction and applications," *Commun. ACM*, vol. 54, no. 9, pp. 69–77, 2011.
[17] B. Dutertre, "Yices 2.2," in *Proc. Int. Conf. Comput. Aided Verification*, 2014, pp. 737–744.
[18] I. Erete and A. Orso, "Optimizing constraint solving to better support symbolic execution," in *Proc. Int. Conf. Softw. Testing Verification Validation*, 2011, pp. 310–315.
[19] S. Jha, R. Limaye, and S. A. Seshia, "Beaver: Engineering an efficient SMT solver for bit-vector arithmetic," in *Proc. Int. Conf. Comput. Aided Verification*, 2009, pp. 668–674.
[20] X. Jia, C. Ghezzi, and S. Ying, "Enhancing reuse of constraint solutions to improve symbolic execution," in *Proc. Int. Symp. Softw. Testing Anal.*, 2015, pp. 177–187.
[21] D. E. Knuth, *The Art of Computer Programming: Combinatorial Algorithms*. Reading, MA, USA: Addison-Wesley, 2011.
[22] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Doklady*, vol. 10, 1966.
[23] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters, "A DPLL(T) theory solver for a theory of strings and regular expressions," in *Proc. Int. Conf. Comput. Aided Verification*, 2014, pp. 646–662.
[24] T. Liu, M. Araujo, M. d'Amorim, and M. Taghdiri, "A comparative study of incremental constraint solving approaches in symbolic execution," in *Proc. Haifa Verification Conf.*, 2014, pp. 284–299.
[25] L. Mariani, M. Pezzè, and M. Santoro, "Gk-Tail+ an efficient approach to learn software models," *IEEE Trans. Software Eng.*, vol. 43, no. 8, pp. 715–738, Aug. 2017.
[26] H. Palikareva and C. Cadar, "Multi-solver support in symbolic execution," in *Proc. Int. Conf. Comput. Aided Verification*, 2013, pp. 53–68.
[27] M. Papalini, K. Khazaei, A. Carzaniga, and D. Rogora, "High throughput forwarding for ICN with descriptors and locators," in *Proc. Symp. Archit. Netw. Commun. Syst.*, 2016, pp. 43–54.
[28] D. M. Perry, A. Mattavelli, X. Zhang, and C. Cadar, "Accelerating array constraints in symbolic execution," in *Proc. Int. Symp. Softw. Testing Anal.*, 2017, pp. 68–78.
[29] M. V. Ramakrishna and J. Zobel, "Performance in practice of string hashing functions," in *Proc. Int. Conf. Database Syst. Adv. Appl.*, 1997, pp. 215–224.
[30] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for JavaScript," in *Proc. IEEE Symp. Secur. Privacy*, 2010, pp. 513–528.
[31] W. Visser, J. Geldenhuys, and M. B. Dwyer, "Green: Reducing, reusing and recycling constraints in program analysis," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2012, pp. 1–11.
[32] S. Wu and U. Manber, "Agrep—A fast approximate pattern-matching tool," in *Proc. USENIX Tech. Conf.*, 1992, pp. 153–162.
[33] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, M. Berzish, J. Dolby, and X. Zhang, "Z3str2: An efficient solver for strings, regular expressions, and length constraints," *Formal Methods Syst. Des.*, vol. 50, no. 2/3, pp. 249–288, 2017.

Andrea Aquino received the master's (honours) degree in computer science from the University of Bologna, Italy, and the PhD degree from the Universitá della Svizzera Italiana (USI), Lugano, Switzerland on distributed caching frameworks for formulas from many logics.

Giovanni Denaro received the PhD degree in computer science and engineering from Politecnico di Milano. He is associate professor of software engineering with the University of Milano-Bicocca. His research interests include software testing and analysis, formal methods for software verification, distributed and service-oriented systems, and software metrics. He has been investigator in several research and development projects in collaboration with leading European universities and companies.

Mauro Pezzè is a professor of software engineering with the University of Milano-Bicocca and with the Universitá della Svizzera Italiana (USI). He is editor in chief of the *ACM Transactions on Software Methodologies*, and has served as an associate editor of the *IEEE Transactions on Software Engineering*, as general chair of the ACM International Symposium on Software Testing and Analysis in 2013, program chair of the International Conference on Software Engineering in 2012 and of the ACM International Symposium on Software Testing and Analysis in 2006. He is known for his work on software testing, program analysis, self-healing, and self-adaptive software systems. He is a senior member of the IEEE.